Arm® Architecture Reference Manual

for R-profile AArch64 architecture

Document number	DDI 0628
Document quality	EAC
Document version	A.a
Document confidentiality	Non-Confidential
Date of issue	27 November 2025

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Document number: DDI 0628

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Release information

The following releases of this document have been made.

Date	Version	Changes
27/Nov/2025	A.a	• Initial release of Non-Confidential AArch64 Arm Architecture Reference Manual for R-profile

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Non-Confidential Proprietary Notice

This document is protected by copyright and other related rights and the use or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm Limited ("Arm"). **No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.**

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether the subject matter of this document infringes any third party patents.

The content of this document is informational only. Any solutions presented herein are subject to changing conditions, information, scope, and data. This document was produced using reasonable efforts based on information available as of the date of issue of this document. The scope of information in this document may exceed that which Arm is required to provide, and such additional information is merely intended to further assist the recipient and does not represent Arm's view of the scope of its obligations. You acknowledge and agree that you possess the necessary expertise in system security and functional safety and that you shall be solely responsible for compliance with all legal, regulatory, safety and security related requirements concerning your products, notwithstanding any information or support that may be provided by Arm herein. In addition, you are responsible for any applications which are used in conjunction with any Arm technology described in this document, and to minimize risks, adequate design and operating safeguards should be provided for by you.

This document may include technical inaccuracies or typographical errors. THIS DOCUMENT IS PROVIDED "AS IS". ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, any patents, copyrights, trade secrets, trademarks, or other rights.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Reference by Arm to any third party's products or services within this document is not an express or implied approval or endorsement of the use thereof.

This document consists solely of commercial items. You shall be responsible for ensuring that any permitted use, duplication, or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word "partner" in reference to Arm's customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of this document shall prevail.

The validity, construction and performance of this notice shall be governed by English Law.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its affiliates) in the US and/or elsewhere. Please follow Arm's trademark usage guidelines at https://www.arm.com/company/policies/trademarks. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

PRE-20349

8 March 2024

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Product Status

The information in this document is final, that is for a developed product.

The information in this manual is at EAC quality, which means that all features of the specification are described in the manual.

Web Address

https://www.arm.com

Limitations of this issue of the Manual

This issue of the Arm Architecture Reference Manual contains many improvements and corrections.

The features implemented by the R-profile architecture are described in R-profile Architecture Extensions, and this manual should be interpreted in accordance with those features. The following sections make reference to features that cannot be implemented within an Armv8-R compliant PE:

- The Ordering requirements defined by the formal concurrency model. This is to ensure that the memory model is conformant with the formal concurrency model.
- The PMU Event Descriptions.

Validation of this document has identified the following issues that Arm will address in future issues:

- Some diagrams in the register descriptions chapter have long field names that split over several lines. These are not wrong, and the descriptions list the field name correctly.
- Arm Pseudocode Definition requires further review and update. Since this appendix is informative, rather than being part of the architecture specification, this does not affect the quality status of this release.
- For a list of the known issues in the most recent Register and Instruction XML releases, please refer to the Release Notes on Developer.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Contents

Arm® Architecture Reference Manual, for R-profile AArch64 architecture

Preface

About this Manual xviii

Using this Manual xxi

Conventions xxvi

Additional reading xxx

Feedback xxxiii

Part A Arm Architecture Introduction and Overview

Chapter A1 Introduction to the Arm Architecture

A1.1 About the Arm architecture 38

A1.2 Architecture profiles 41

A1.3 Arm architectural concepts 43

A1.4 Supported data types 47

A1.5 Floating-point support 56

A1.6 The Arm memory model 67

A1.7 Reliability, Availability, and Serviceability 69

Chapter A2 R-profile Architecture Extensions

A2.1 About the Armv8-R AArch64 architecture extensions 74

A2.2 Armv8-R AArch64 architecture extensions 76

Part B The AArch64 Application Level Architecture

Chapter B1 The AArch64 Application Level Programmers' Model

B1.1 About the Application level programmers' model 111

B1.2 Registers in AArch64 Execution state 113

B1.3 Process state, PSTATE 117

B1.4 Software control features and EL0 120

B1.5 Memory barriers 124

B1.6 Caches and memory hierarchy 132

B1.7 Alignment support 138

B1.8 Endian support 142

B1.9 Memory types and attributes 146

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B1.10 Mismatched memory attributes 158
B1.11 Synchronization and semaphores 162

Part C The AArch64 Instruction Set

Chapter C1 The A64 Instruction Set

C1.1 About the A64 instruction set 176
C1.2 Structure of the A64 assembler language 178
C1.3 Address generation 185
C1.4 Instruction aliases 189

Chapter C2 About the A64 Instruction Descriptions

C2.1 Understanding the A64 instruction descriptions 192
C2.2 General information about the A64 instruction descriptions 196

Chapter C3 A64 Instruction Set Overview

C3.1 Branches, Exception generating, and System instructions 201
C3.2 Loads and stores 213
C3.3 Data processing - immediate 230
C3.4 Data processing - register 236
C3.5 Data processing - SIMD and floating-point 246

Chapter C4 A64 Instruction Set Encoding

C4.1 A64 instruction set encoding 272

Chapter C5 The A64 System Instruction Class

C5.1 The System instruction class encoding space 358
C5.2 Special-purpose registers 370
C5.3 A64 System instructions for cache maintenance 418
C5.4 A64 System instructions for address translation 444
C5.5 A64 System instructions for TLB maintenance 460
C5.6 A64 System instructions for prediction restriction 651
C5.7 A64 IMPLEMENTATION DEFINED System instructions 662

Chapter C6 A64 Base Instruction Descriptions

C6.1 About the A64 base instructions 667
C6.2 Alphabetical list of A64 base instructions 671

Chapter C7 A64 Advanced SIMD and Floating-point Instruction Descriptions

C7.1 About the A64 Advanced SIMD and floating-point instructions 1169
C7.2 Alphabetical list of A64 Advanced SIMD and floating-point instructions 1173

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Part D The AArch64 System Level Architecture

Chapter D1 The AArch64 System Level Programmers' Model

D1.1 Exception levels 2037
D1.2 Registers for instruction processing and exception handling 2040
D1.3 Exceptions 2043
D1.4 Process state, PSTATE 2065
D1.5 Resets and power domains 2068
D1.6 Mechanisms for entering a low-power state 2072
D1.7 Self-hosted debug 2078
D1.8 Event monitors 2081

Chapter D2 AArch64 Self-hosted Debug

D2.1 About self-hosted debug 2084
D2.2 Routing debug exceptions 2088
D2.3 The debug exception enable controls 2090
D2.4 The effect of powerdown on debug exceptions 2093
D2.5 Summary of the routing and enabling of debug exceptions 2095
D2.6 Pseudocode description of debug exceptions 2098
D2.7 Breakpoint Instruction exceptions 2100
D2.8 Breakpoint exceptions 2103
D2.9 Watchpoint exceptions 2119
D2.10 Software Step exceptions 2135
D2.11 Synchronization and debug exceptions 2147

Chapter D3 AArch64 Self-hosted Trace

D3.1 About self-hosted trace 2150
D3.2 Prohibited regions in self-hosted trace 2152
D3.3 Self-hosted trace timestamps 2155
D3.4 Synchronization in self-hosted trace 2157

Chapter D4 The AArch64 System Level Memory Model

D4.1 About the memory system architecture 2160
D4.2 Address space 2162
D4.3 Mixed-endian support in AArch64 2164
D4.4 Cache support 2166
D4.5 External aborts 2191
D4.6 Memory barrier instructions 2194
D4.7 Pseudocode description of general memory System instructions 2196

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter D5 **The AArch64 Virtual Memory System Architecture**

D5.1 Address translation 2201
D5.2 Translation process 2210
D5.3 Translation table descriptor formats 2228
D5.4 Memory access control 2237
D5.5 Hardware updates to the translation tables 2244
D5.6 Memory region attributes 2252
D5.7 Other descriptor fields 2255
D5.8 Address tagging 2257
D5.9 Pointer authentication 2259
D5.10 Memory aborts 2265
D5.11 Translation Lookaside Buffers 2276
D5.12 TLB maintenance 2281
D5.13 Caches 2295
D5.14 Pseudocode description of VMSA address translation 2298

Chapter D6 **The AArch64 Protected Memory System Architecture**

D6.1 About the Protected Memory System Architecture 2302
D6.2 Memory Protection Unit 2304
D6.3 Address translation regimes 2306
D6.4 Default memory map 2308
D6.5 MPU memory translations and faults 2310
D6.6 Protection region attributes and access permissions 2319
D6.7 MPU fault encodings 2325
D6.8 PMSAv8-64 implications for caches 2327
D6.9 Address tagging and pointer authentication support 2329
D6.10 Security model 2331
D6.11 Virtualization 2334

Chapter D7 **The Generic Timer in AArch64 state**

D7.1 About the Generic Timer 2337
D7.2 The AArch64 view of the Generic Timer 2342

Chapter D8 **The Performance Monitors Extension**

D8.1 About the Performance Monitors 2348
D8.2 Accuracy of the Performance Monitors 2352
D8.3 Behavior on overflow 2355
D8.4 Attributability 2358
D8.5 Controlling the PMU counters 2360

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.6 Multithreaded implementations 2364
D8.7 Event filtering 2366
D8.8 Performance Monitors and Debug state 2369
D8.9 Counter access 2371
D8.10 Performance Monitors Extension registers 2374

Chapter D9 PMU Event Descriptions

D9.1 Definitions 2377
D9.2 The PMU event number space and common events 2387
D9.3 Common event numbers 2432
D9.4 Cycle event counting 2581
D9.5 Meaningful combinations of common events 2584
D9.6 Required events 2590
D9.7 Synchronous events 2593
D9.8 IMPLEMENTATION DEFINED event numbers 2595

Chapter D10 RAS PE Architecture

D10.1 About the RAS Extension 2598
D10.2 PE error handling 2600
D10.3 Generating error exceptions 2606
D10.4 Taking error exceptions 2608
D10.5 Error synchronization event 2617
D10.6 Virtual SError exceptions and delegated SError exceptions 2623
D10.7 Error records in the PE 2625

Chapter D11 AArch64 System Register Encoding

D11.1 The System register encoding space 2630
D11.2 Moves to and from debug and trace System registers 2632
D11.3 Moves to and from non-debug System registers, Special-purpose registers 2635

Chapter D12 AArch64 System Register Descriptions

D12.1 About the AArch64 System registers 2646
D12.2 General system control registers 2657
D12.3 Debug registers 2981
D12.4 Trace registers 3052
D12.5 Performance Monitors registers 3060
D12.6 PMSA registers 3116
D12.7 RAS registers 3140
D12.8 Generic Timer registers 3174

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Part E External Debug

Chapter E1 About External Debug

E1.1 Introduction to external debug 3212
E1.2 External debug 3214
E1.3 Required debug authentication 3216

Chapter E2 Debug State

E2.1 About Debug state 3219
E2.2 Halting the PE on debug events 3221
E2.3 Entering Debug state 3228
E2.4 Behavior in Debug state 3232
E2.5 Exiting Debug state 3252

Chapter E3 Halting Debug Events

E3.1 Introduction to Halting debug events 3256
E3.2 Halting Step debug events 3259
E3.3 Halt Instruction debug event 3269
E3.4 Exception Catch debug event 3271
E3.5 External Debug Request debug event 3275
E3.6 OS Unlock Catch debug event 3278
E3.7 Reset Catch debug events 3280
E3.8 Software Access debug event 3282
E3.9 Synchronization and Halting debug events 3284

Chapter E4 The Debug Communication Channel and Instruction Transfer Register

E4.1 Introduction 3287
E4.2 DCC and ITR registers 3289
E4.3 DCC and ITR access modes 3292
E4.4 Flow control of the DCC and ITR registers 3297
E4.5 Synchronization of DCC and ITR accesses 3302
E4.6 Interrupt-driven use of the DCC 3309
E4.7 Pseudocode description of the operation of the DCC and ITR registers 3311

Chapter E5 The Embedded Cross-Trigger Interface

E5.1 About the Embedded Cross-Trigger 3314
E5.2 Basic operation on the ECT 3317
E5.3 Cross-triggers on a PE in an Arm R-profile implementation 3322
E5.4 Description and allocation of CTI triggers 3324
E5.5 CTI registers programmers' model 3329

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E5.6 Examples 3331

Chapter E6 Debug Reset and Powerdown Support

E6.1 About Debug over powerdown 3336
E6.2 Power domains and debug 3338
E6.3 Core power domain power states 3340
E6.4 Powerup request mechanism 3343
E6.5 Emulating low-power states 3346
E6.6 Debug OS Save and Restore sequences 3349
E6.7 Reset and debug 3356

Chapter E7 The PC Sample-based Profiling Extension

E7.1 About the PC Sample-based Profiling Extension 3360

Chapter E8 About the External Debug Registers

E8.1 Relationship between external debug and System registers 3365
E8.2 Endianness and supported access sizes 3367
E8.3 Synchronization of changes to the external debug registers 3369
E8.4 Memory-mapped accesses to the external debug interface 3374
E8.5 External debug interface register access permissions 3377
E8.6 External debug interface registers 3381
E8.7 Cross-trigger interface registers 3387
E8.8 External debug register resets 3390

Chapter E9 External Debug Register Descriptions

E9.1 About the external debug registers 3395
E9.2 External debug registers 3397
E9.3 External trace registers 3503
E9.4 Cross-Trigger Interface registers 3505

Part F Memory-mapped Components of the Arm Architecture

Chapter F1 Requirements for Memory-mapped Components

F1.1 Supported access sizes 3566
F1.2 Synchronization of memory-mapped registers 3569
F1.3 Access requirements for reserved and unallocated registers 3572

Chapter F2 System Level Implementation of the Generic Timer

F2.1 About the Generic Timer specification 3575
F2.2 Memory-mapped counter module 3578
F2.3 Memory-mapped timer components 3583

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter F3 **Recommended External Interface to the Performance Monitors**

F3.1 About the external interface to the Performance Monitors registers 3589

Chapter F4 **External System Control Register Descriptions**

F4.1 About the external system control register descriptions 3595

F4.2 External Performance Monitors registers summary 3597

F4.3 Performance Monitors external register descriptions 3601

F4.4 Generic Timer memory-mapped registers overview....3678

F4.5 Generic Timer memory-mapped register descriptions 3680

Part G Architectural Pseudocode

Chapter G1 **R-profile Architecture Pseudocode**

G1.1 Pseudocode for AArch64 operation 3731

G1.2 Shared pseudocode....3863

Part H Appendixes

Appendix H1 **Architectural Constraints on UNPREDICTABLE Behaviors**

H1.1 AArch64 CONSTRAINED UNPREDICTABLE behaviors....4032

Appendix H2 **Recommendations for Reporting Memory Attributes on an Interconnect**

H2.1 Arm recommendations for reporting memory attributes on an interconnect 4051

Appendix H3 **Arm Pseudocode Definition**

H3.1 About the Arm pseudocode....4054

H3.2 Pseudocode for instruction descriptions....4056

H3.3 Data types 4059

H3.4 Operators 4065

H3.5 Statements and control structures....4072

H3.6 Built-in functions 4079

H3.7 Miscellaneous helper procedures and functions 4083

H3.8 Arm pseudocode definition index 4087

Appendix H4 **Registers Index**

H4.1 Introduction and register disambiguation....4092

H4.2 Alphabetical index of AArch64 registers and System instructions 4095

H4.3 Functional index of AArch64 registers and System instructions 4106

H4.4 Alphabetical index of memory-mapped registers 4117

H4.5 Functional index of memory-mapped registers....4125

Glossary

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Contents

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Preface

This preface introduces the *Arm Architecture Reference Manual, for A-profile architecture*. It contains the following sections:

- About this Manual.
- Using this Manual.
- Conventions.
- Additional reading.
- Feedback.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

About this Manual

This Manual describes the Arm® architecture v8, Armv8. The architecture describes the operation of an Armv8-R AArch64 Processing element (PE), and this Manual includes descriptions of:

- The AArch64 Execution state.
- The A64 instruction set for AArch64 state.
- The Exception model.
- The memory model, that defines memory ordering and memory management. This Manual covers the Arm R architecture profile for AArch64 state, that defines a *Virtual Memory System Architecture* (VMSA) and a *Protected Memory System Architecture* (PMSA).
- The programmers' model, and its interfaces to System registers that control most PE and memory system features, and provide status information.
- The Advanced SIMD and floating-point instructions, which provide high-performance:
  - Single-precision, half-precision, and double-precision floating-point operations.
  - Conversions between double-precision, single-precision, and half-precision floating-point values.
  - Integer, single-precision floating-point, and half-precision floating-point vector operations in all instruction sets.
  - Double-precision floating-point vector operations in the A64 instruction set.
- The security model, which provides up to four Security states to support Secure applications and confidential computing.
- The virtualization model.
- The Debug architecture, which provides software access to debug features.

This Manual gives the assembler syntax for the instructions it describes, meaning that it describes instructions in textual form. However, this Manual is not a tutorial for Arm assembler language, nor does it describe Arm assembler language, except at a basic level. To make effective use of Arm assembler language, read the documentation supplied with the assembler being used.

This Manual is organized into parts:

Part A Provides an introduction to the Arm architecture, and an overview of the AArch64 state.

Part B Describes the application level view of the AArch64 Execution state, meaning the view from EL0. It describes the application level view of the programmers' model and the memory model.

Part C Describes the A64 instruction set, which is available in the AArch64 Execution state. The descriptions for each instruction also include the precise effects of each instruction when executed at EL0, described as *unprivileged* execution, including any restrictions on its use, and how the effects of the instruction differ at higher Exception levels. This information is of primary importance to authors and users of compilers, assemblers, and other programs that generate Arm machine code.

Part D Describes the system level view of the AArch64 Execution state. It includes details of the System registers, most of which are not accessible from EL0, and the system level view of the programmers' model and the memory model. This part includes the description of self-hosted debug.

Part E Describes the Debug architecture for external debug. This provides configuration, breakpoint and watchpoint support, and a *Debug Communications Channel* (DCC) to a debug host.

Part F Describes additional features of the architecture that are not closely coupled to a PE, and therefore are accessed through memory-mapped interfaces. Some of these features are OPTIONAL.

Part G Provides pseudocode that describes various features of the Arm architecture.

Part H, Appendixes

Provide additional information.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Glossary Defines terms used in this Manual that have a specialized meaning.

Note

Terms that are generally well understood in the microelectronics industry are not included in the Glossary.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Using this Manual

The information in this Manual is organized into parts, as described in this section.

Part A, Introduction and Architecture Overview

Part A gives an overview of the Armv8-R for AArch64 architecture profile, including its relationship to the other Arm PE architectures. It introduces the terminology used to describe the architecture.

It contains the following chapters:

Introduction to the Arm Architecture

Read this for an introduction to the Arm architecture.

R-profile Architecture Extensions

Read this for an introduction to the R-profile architecture extensions and features.

Part B, The AArch64 Application Level Architecture

Part B describes the AArch64 state application level view of the architecture. It contains the following chapters:

The AArch64 Application Level Programmers' Model

Read this for an application level description of the programmers' model for software executing in AArch64 state. It describes execution at EL0 when EL0 is using AArch64 state.

The AArch64 Application Level Memory Model

Read this for an application level description of the memory model for software executing in AArch64 state. It describes the memory model for execution in EL0 when EL0 is using AArch64 state. It includes information about Arm memory types, attributes, and memory access controls.

Part C, The A64 Instruction Set

Part C describes the A64 instruction set, which is used in AArch64 state. It contains the following chapters:

The A64 Instruction Set

Read this for a description of the A64 instruction set and common instruction operation details.

About the A64 Instruction Descriptions

Read this to understand the format of the A64 instruction descriptions.

A64 Instruction Set Overview

Read this for an overview of the A64 instructions.

A64 Instruction Set Encoding

Read this for a description of the A64 instruction set encoding.

The A64 System Instruction Class

Read this for a description of the AArch64 System instructions and register descriptions, and the System instruction class encoding space.

A64 Base Instruction Descriptions

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Read this for information on key aspects of the A64 base instructions and for descriptions of the individual instructions, which are listed in alphabetical order.

A64 Advanced SIMD and Floating-point Instruction Descriptions

Read this for information on key aspects of the A64 Advanced SIMD and floating-point instructions and for descriptions of the individual instructions, which are listed in alphabetical order.

Part D, The AArch64 System Level Architecture

Part D describes the AArch64 state system level view of the architecture. It contains the following chapters:

The AArch64 System Level Programmers' Model

Read this for a description of the AArch64 state system level view of the programmers' model.

AArch64 Self-hosted Debug

Read this for an introduction to, and a description of, self-hosted debug in AArch64 state.

AArch64 Self-hosted Trace

Read this for an introduction to, and a description of, self-hosted trace in AArch64 state.

The AArch64 System Level Memory Model

Read this for a description of the AArch64 state system level view of the general features of the memory system.

The AArch64 Virtual Memory System Architecture

Read this for a system level view of the AArch64 *Virtual Memory System Architecture* (VMSA), the memory system architecture of an R-profile architecture implementation executing in AArch64 state.

The AArch64 Protected Memory System Architecture

Read this for a system level view of the AArch64 *Protected Memory System Architecture* (PMSA), the memory system architecture of an R-profile architecture implementation executing in AArch64 state.

The Generic Timer in AArch64 state

Read this for a description of the AArch64 view of the Arm Generic Timer.

The Performance Monitors Extension

Read this for a description of the Arm Performance Monitors, an optional non-invasive debug component.

RAS PE Architecture

Read this for a description of the RAS Extension PE architecture.

AArch64 System Register Encoding

Read this for a description of the encoding of the AArch64 System registers, and the other uses of the AArch64 System registers encoding space.

AArch64 System Register Descriptions

Read this for an introduction to, and description of, each of the AArch64 System registers.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Part E, External Debug

Part E describes the architecture for external debug. It contains the following chapters:

About External Debug

Read this for an introduction to external debug, and a definition of the scope of this part of the Manual.

Debug State

Read this for a description of Debug state, which the PE might enter as the result of a Halting debug event.

Halting Debug Events

Read this for a description of the external debug events referred to as Halting debug events.

The Debug Communication Channel and Instruction Transfer Register

Read this for a description of the communication between a debugger and the PE debug logic using the Debug Communications Channel and the Instruction Transfer register.

The Embedded Cross-Trigger Interface

Read this for a description of the Embedded Cross-Trigger Interface.

Debug Reset and Powerdown Support

Read this for a description of reset and powerdown support in the Debug architecture.

The PC Sample-based Profiling Extension

Read this for a description of the PC Sample-based Profiling Extension that is an OPTIONAL extension to an Armv8 or Armv9 implementation.

About the External Debug Registers

Read this for some additional information about the external debug registers.

External Debug Register Descriptions

Read this for a description of each external debug register.

Part F, Memory-mapped Components of the Arm architecture

Part F describes the memory-mapped components in the architecture. It contains the following chapters:

Requirements for Memory-mapped Components

Read this for descriptions of some general requirements for memory-mapped components within a system that complies with the Arm architecture.

System Level Implementation of the Generic Timer

Read this for a definition of a system level implementation of the Generic Timer.

Recommended External Interface to the Performance Monitors

Read this for a description of the recommended memory-mapped and external debug interfaces to the Performance Monitors.

External System Control Register Descriptions

Read this for a description of each memory-mapped system control register.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Part G, Architectural Pseudocode

Part G contains pseudocode that describes various features of the Arm architecture. It contains the following chapter:

R-profile Architecture Pseudocode

Read this for the pseudocode definitions that describe various features of the Arm architecture, for operation in AArch64 state.

Part H, Appendixes

Note

Some of the descriptions in the following appendixes are not part of the Arm architecture specification. They are included here as supplementary information, for the convenience of developers and users who might require this information.

This Manual contains the following appendixes:

Architectural Constraints on UNPREDICTABLE Behaviors

Read this for a description of the architecturally-required constraints on UNPREDICTABLE behaviors in the Arm architecture.

Arm Pseudocode Definition

Read this for definitions of the Arm pseudocode.

Registers Index

Read this for an alphabetic and functional index of AArch64 registers, and memory-mapped registers.

Glossary

Defines terms used in this Manual that have a specialized meaning.

Note

Terms that are well understood in the microelectronics industry are not included in the Glossary.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Conventions

The following sections describe conventions that this book can use:

- Typographic conventions.
- Rules-based writing.
- Signals.
- Numbers.
- Pseudocode descriptions.
- Assembler syntax descriptions.

Typographic conventions

The typographical conventions are:

italic Introduces special terminology, and denotes citations.

bold Denotes signal names, and is used for terms in descriptive lists, where appropriate.

`monospace` Used for assembler syntax descriptions, pseudocode, and source code examples.

Also used in the main text for instruction mnemonics and for references to other items appearing in assembler syntax descriptions, pseudocode, and source code examples.

SMALL CAPITALS

Used in body text for a few terms that have specific technical meanings, and are defined in the Glossary.

Colored text Indicates a link. This can be:

- A URL, for example https://developer.arm.com.
- A cross-reference that includes the page number of the referenced information if it is not on the current page, for example, Assembler syntax descriptions.
- A link, to a chapter or appendix, or to a glossary entry, or to the section of the Manual that defines the colored term, for example Simple sequential execution or SCTLR.

{ and } Braces, { and }, have two distinct uses:

Optional items

In syntax descriptions braces enclose optional items. In the following example they indicate that the `<shift>`parameter is optional:

```
ADD <Wd|WSP>, <Wn|WSP>, #<imm>{, <shift>}
```

Similarly, they can be used in generalized field descriptions, for example TCR_ELx.{I}PS refers to a field in the TCR_ELx registers that is called either IPS or PS.

Sets of items

Braces can be used to enclose sets. For example, HCR_EL2.{E2H, TGE} refers to a set of two register fields, HCR_EL2.E2H and HCR_EL2.TGE.

Notes Notes are formatted as:

Note

This is a Note.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

In this Manual, Notes are used only to provide additional information, usually to help understanding of the text. While a Note might repeat architectural information given elsewhere in the Manual, a Note never provides any part of the definition of the architecture.

Rules-based writing

Some sections of this Manual use rules-based writing. Rules-based writing consists of a set of individual content items. A content item is classified as one of the following:

- Rule.
- Information.
- Software usage.
- Declaration.

Rules are normative statements. An implementation that is compliant with this specification must conform to all Rules in this Manual that apply to that implementation.

Rules must not be read in isolation. Where a particular feature is specified by multiple Rules, these are generally grouped into sections and subsections that provide context. Where appropriate, these sections begin with a short introduction.

Arm strongly recommends that implementers read all chapters and sections of this Manual to ensure that an implementation is compliant.

Content items other than Rules are informative statements. These are provided as an aid to understanding this Manual.

Content item identifiers

A content item may have an associated identifier which is unique among content items in this Manual. After content reaches beta status, a given content item has the same identifier across subsequent versions of this Manual.

Content item rendering

In this Manual, a content item is rendered with a token of the following format in the left margin: L_{iiii}. L is a label that indicates the content class of the content item. iiii is the identifier of the content item.

Content item classes

Each of the content item classes has a different function in this Manual.

Rule

A Rule is a statement that describes the behavior of a compliant implementation.

A Rule explains what happens in a particular situation.

A Rule does not define concepts or terminology.

A Rule is rendered with the label R.

Information

An Information statement provides information and guidance as an aid to understanding the Manual.

An Information statement is rendered with the label I.

Software usage

A Software usage statement provides guidance on how software can make use of the features defined by the specification.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A Software usage statement is rendered with the label S.

Declaration

A Declaration statement introduces concepts or terminology.

A Declaration does not describe behavior.

A Declaration is rendered with the label D.

Signals

In general this specification does not define hardware signals, but it does include some signal examples and recommendations. The signal conventions are:

Signal level The level of an asserted signal depends on whether the signal is active-HIGH or active-LOW. Asserted means:

- HIGH for active-HIGH signals.
- LOW for active-LOW signals.

Lowercase n At the start or end of a signal name denotes an active-LOW signal.

Numbers

Numbers are normally written in decimal. Binary numbers are preceded by `0b`, and hexadecimal numbers by `0x`. In both cases, the prefix and the associated value are written in a `monospace` font, for example `0xFFFF_0000`. To improve readability, long numbers can be written with an underscore separator between every four characters, for example `0xFFFF_0000_0000_0000`. Ignore any underscores when interpreting the value of a number.

Pseudocode descriptions

This Manual uses a form of pseudocode to provide precise descriptions of the specified functionality. This pseudocode is written in `monospace` font, and is described in Arm Pseudocode Definition.

Assembler syntax descriptions

This Manual contains numerous syntax descriptions for assembler instructions and for components of assembler instructions. These are shown in a `monospace` font, and use the conventions described in Structure of the A64 assembler language, and Arm Pseudocode Definition.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Additional reading

This section lists relevant publications from Arm and third parties.

See Arm Developer, https://developer.arm.com, for access to Arm documentation.

Arm publications

- *AMBA® 4 ATB Protocol Specification, ATBv1.0 and ATBv1.1* (ARM IHI 0032).
- *Arm® Architecture Reference Manual, for A-profile architecture*, (ARM DDI 0487)
- *ARM® Architecture Reference Manual, ARMv7-A and ARMv7-R edition* (ARM DDI 0406).
- *Arm® Memory System Resource Partitioning and Monitoring (MPAM) System Component Specification* (ARM IHI 0099).
- *Arm® Architecture Reference Manual Supplement - Armv8, for Armv8-R AArch64 architecture profile* (ARM DDI 0600).
- *ARM® Architecture Reference Manual Supplement - Armv8, for the ARMv8-R AArch32 architecture profile* (ARM DDI 0568).
- *Arm® Architecture Reference Manual Supplement, Transactional Memory Extension (TME), for A-profile architecture* (ARM DDI 0617).
- *Arm® Debug Interface Architecture Specification, ADIv6.0* (ARM IHI 0074).
- *Arm® Debug Interface Architecture Specification, ADIv5.0 to ADIv5.2* (ARM IHI 0031).
- *Arm® Embedded Trace Macrocell Architecture Specification, ETMv4* (ARM IHI 0064).
- *Arm® Generic Interrupt Controller Architecture Specification, GIC architecture version 3.0 and version 4.0* (ARM IHI 0069).
- *ARM® CoreSight™ SoC Technical Reference Manual* (ARM DDI 0480).
- *Arm® CoreSight™ Architecture Specification* (ARM IHI 0029).
- *ARM® Procedure Call Standard for the ARM 64-bit Architecture* (ARM IHI 0055).
- *Arm® Reliability, Availability, and Serviceability (RAS) System Architecture, for A-profile architecture* (ARM IHI 0100).
- *Arm® Server Base System Architecture* (ARM DEN 0029).
- *Arm® Base System Architecture* (ARM DEN 0094).

Other publications

The following publications are referred to in this Manual, or provide more information:

- *Announcing the Advanced Encryption Standard (AES)*, Federal Information Processing Standards Publication 197, November 2001.
- IEEE Std 754-2008, *IEEE Standard for Floating-point Arithmetic*, August 2008.
- IEEE Std 754-1985, *IEEE Standard for Floating-point Arithmetic*, March 1985.
- *Secure Hash Standard (SHA)*, Federal Information Processing Standards Publication 180-2, August 2002.
- *The Galois/Counter Mode of Operation*, McGraw, D. and Viega, J., Submission to NIST Modes of Operation Process, January 2004.
- *Memory Consistency Models for Shared Memory-Multiprocessors*, Gharachorloo, Kourosh, 1995, Stanford University Technical Report CSL-TR-95-685.
- *Standard Manufacturer's Identification Code, JEP106*, JEDEC Solid State Technology Association.
- *SM3 Cryptographic Hash Algorithm*, China Internet Network Information Center (CNNIC).

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- *SM4 Block Cipher Algorithm*, China Internet Network Information Center (CNNIC).
- *The QARMA Block Cipher Family*, Roberto Avanzi, Qualcomm Product Security Initiative. Available from https://eprint.iacr.org/2016/444.
- *OCP 8-bit Floating Point Specification (OFP8)* Paulius Micikevicius, Stuart Oberman, Pradeep Dubey, Marius Cornea, Andres Rodriguez, Ian Bratt, Richard Grisenthwaite, et al., June 2023. Available from https://www.opencompute.org/documents/ocp-8-bit-floating-point-specification-ofp8-revision-1-0-2023-12-01-pdf-1

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Feedback

Arm welcomes feedback on its documentation.

Feedback on this Manual

If you have any comments or queries about this Manual, create a ticket at https://support.developer.arm.com.

As part of the ticket, include:

- The title, Arm® Architecture Reference Manual, for R-profile AArch64 architecture.
- The number, DDI 0628 A.a.
- The section name to which your comments refer.
- The page number(s) to which your comments refer.
- The rule identifier(s) to which your comments refer, if applicable.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Note

Arm tests PDFs only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the appearance or behavior of any document when viewed with any other PDF reader. Search performance may be significantly improved by increasing the Fast Find Maximum Cache Size under Search in Preferences.
If using Google Chrome™ or Microsoft Edge™, load time is significantly improved since the April 2025 release of these browsers.
Users may find, on some systems, that PDF viewers such as Okular or Foxit™ give good performance.

Inclusive terminology commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used terms that can be offensive. Arm strives to lead the industry and create change.

Previous issues of this document included language that can be offensive. We have replaced this language. To report offensive language in this document, email `terms@arm.com`.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Part A
Arm Architecture Introduction and Overview

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter A1
Introduction to the Arm Architecture

This chapter introduces the Arm architecture. It contains the following sections:

- About the Arm architecture.
- Architecture profiles.
- Arm architectural concepts.
- Supported data types.
- Floating-point support.
- The Arm memory model.
- Reliability, Availability, and Serviceability.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A1.1 About the Arm architecture

The Arm architecture described in this Architecture Reference Manual defines the behavior of an abstract machine, referred to as a *processing element*, often abbreviated to *PE*. Implementations compliant with the Arm architecture must conform to the described behavior of the processing element. It is not intended to describe how to build an implementation of the PE, nor to limit the scope of such implementations beyond the defined behaviors.

Except where the architecture specifies differently, the programmer-visible behavior of an implementation that is compliant with the Arm architecture must be the same as a simple sequential execution of the program on the processing element. The programmer-visible behavior does not include a specified execution time for any instruction but requires each instruction to execute in finite time, as long as another observer is not continually modifying any Location associated with a Memory Effect, other than an Explicit Memory Effect, that is architecturally required for execution of the instruction. In cases where another observer is continually modifying a Location associated with a Memory Effect, other than an Explicit Memory Effect, at least one of the observers must make forward progress in finite time. Unless otherwise specified, the architecturally defined effects of each instruction also complete in finite time.

The Arm Architecture Reference Manual also describes rules for software to use the PE.

The Arm architecture includes definitions of:

- An associated debug architecture, see:
  - AArch64 Self-hosted Debug.
  - AArch32 Self-hosted Debug, AArch32 state is not described in this Manual.
  - External Debug.
- Associated trace architectures that define trace units that implementers can implement with the associated processor hardware. For more information, see:
  - The *Arm® Embedded Trace Macrocell Architecture Specification, ETMv4* (ARM IHI 0064).
  - AArch64 Self-hosted Trace.

Arm also provides system architecture specifications for systems aimed at particular product categories, such as servers, or personal computers, *Arm® Server Base System Architecture* (ARM DEN 0029) and *Arm® Base System Architecture* (ARM DEN 0094). These specifications build on the architecture specification in this Manual, restricting, as defined in those specifications, some of the optional and IMPLEMENTATION DEFINED features defined in this specification, as well as specifying additional features applicable at the system level.

The Arm architecture is a *Reduced Instruction Set Computer* (RISC) architecture with the following RISC architecture features:

- A large uniform register file.
- A *load/store* architecture, where data-processing operations only operate on register contents, not directly on memory contents.
- Simple addressing modes, with all load/store addresses determined from register contents and instruction fields only.

The architecture defines the interaction of the PE with memory, including caches, and includes a memory translation system. It also describes how multiple PEs interact with each other and with other observers in a system.

This document defines version 8of the R-profile architecture.

See Architecture profiles for more information.

The Arm architecture supports implementations across a wide range of performance points. Implementation size, performance, and very low power consumption are key attributes of the Arm architecture.

An important feature of the Arm architecture is backward compatibility, combined with the freedom for optimal implementation in a wide range of standard and more specialized use cases.

Features that are optional are explicitly defined as such in this Manual.

Note

The presence of an ID register field for a feature does not imply that the feature is optional.

Both Execution states support floating-point instructions:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- AArch64 state provides:
  - Advanced SIMD instructions that operate on registers in the SIMD&FP register file.
  - Scalar floating-point instructions that operate on registers in the SIMD&FP register file.

    Note

    The A64 instruction set does not include SIMD instructions that operate on the general-purpose registers, therefore, some AArch64 instructions descriptions use SIMD as a synonym for Advanced SIMD.

Note

See Conventions for information about conventions used in this Manual, including the use of SMALL CAPITALS for particular terms that have Arm-specific meanings that are defined in the Glossary.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A1.2 Architecture profiles

The Arm architecture has evolved significantly since its introduction, and Arm continues to develop it. Nine major versions of the architecture have been defined to date, denoted by the version numbers 1 to 9. Of these, the first three versions are now obsolete.

The generic names AArch64 and AArch32 describe the 64-bit and 32-bit Execution states:

AArch64 Is the 64-bit Execution state, meaning addresses are held in 64-bit registers, and instructions in the base instruction set can use 64-bit registers for their processing. AArch64 state supports the A64 instruction set.

AArch32 Is the 32-bit Execution state, meaning addresses are held in 32-bit registers, and instructions in the base instruction sets use 32-bit registers for their processing. AArch32 state supports the T32 and A32 instruction sets. AArch32 is not described in this Manual.

Note

The *Base instruction set* comprises the supported instructions other than the floating-point instructions.

See sections Execution state and The instruction sets for more information.

Arm defines three architecture profiles:

A Application profile, described in this Manual:

- Supports a *Virtual Memory System Architecture* (VMSA) based on a *Memory Management Unit* (MMU).

  Note

  An Armv8-A implementation can be called an AArchv8-A implementation and an Armv9-A implementation can be called an AArchv9-A implementation.

- Supports the A64, A32, and T32 instruction sets. The T32 and A32 instruction sets are not described in this Manual.

R Real-time profile:

- Supports a *Protected Memory System Architecture* (PMSA) based on a *Memory Protection Unit* (MPU).
- Supports an optional VMSA based on an MMU.
- Supports the A64, or the A32 and T32 instruction sets.

M Microcontroller profile:

- Implements a programmers' model designed for low-latency interrupt processing, with hardware stacking of registers and support for writing interrupt handlers in high-level languages.
- Supports a PMSA based on an MPU.
- Supports a variant of the T32 instruction set.

This Manual describes only Armv8-R AArch64. For information about the A and M architecture profiles, and earlier Arm architecture versions, see:

- The *Arm® Architecture Reference Manual, for A-profile architecture*.
- The *ARM® Architecture Reference Manual Supplement - ARMv8, for the ARMv8-R AArch32 architecture profile.*
- The *ARM® Architecture Reference Manual ARMv7-A and ARMv7-R edition*.
- The *Armv8-M Architecture Reference Manual*.
- The *Armv7-M Architecture Reference Manual*.
- The *ARMv6-M Architecture Reference Manual*.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A1.3 Arm architectural concepts

This section introduces architectural concepts and the associated terminology.

The following subsections describe key architectural concepts. Each section introduces the corresponding terms that are used to describe the architecture:

- Execution state.
- The instruction sets.
- System registers.
- Arm Debug.

A1.3.1 Execution state

The Execution state defines the PE execution environment, including:

- The supported register widths.
- The supported instruction sets.
- Significant aspects of:
  - The Exception model.
  - The *Virtual Memory System Architecture* (VMSA).
  - The programmers' model.

The Execution states are:

AArch64 The 64-bit Execution state. This Execution state:

- Provides 31 64-bit general-purpose registers, of which X30 is used as the procedure link register.
- Provides a 64-bit *Program Counter* (PC), *stack pointers* (SPs), and *Exception Link Registers* (ELRs).
- Provides 32 128-bit registers for Advanced SIMD vector and scalar floating-point support.
- Provides a single instruction set, A64. For more information, see The instruction sets.
- Defines the Armv8 Exception model, with three Exception levels, EL0 to EL2 that provide an *execution privilege* hierarchy, see Exception levels.
- Provides support for 64-bit *virtual addressing*. For more information, including the limits on address ranges, see The AArch64 Virtual Memory System Architecture.
- Defines a number of *Process state* (PSTATE) elements that hold PE state. The A64 instruction set includes instructions that operate directly on various PSTATE elements.
- Names each System register using a suffix that indicates the lowest Exception level at which the register can be accessed.

AArch32 AArch32 state is not described in this Manual.

A1.3.2 The instruction sets

The possible instruction sets depend on the Execution state:

AArch64 AArch64 state supports a single instruction set, called A64. This is a fixed-length instruction set that uses 32-bit instruction encodings.

For information on the A64 instruction set, see A64 Instruction Set Overview.

The instruction sets support SIMD and scalar floating-point instructions. See Floating-point support.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A1.3.3 System registers

System registers provide control and status information of architected features.

The System registers use a standard naming format: <register_name>.<bit_field_name> to identify specific registers as well as control and status bits within a register.

Bits can also be described by their numerical position in the form <register_name>[x:y] or the generic form bits[x:y].

In addition, in AArch64 state, most register names include the lowest Exception level that can access the register as a suffix to the register name:

- <register_name>_ELx, where x is 0, 1, or 2.

For information about Exception levels, see Exception levels.

The System registers comprise:

- The following registers that are described in this Manual:
  - General system control registers.
  - Debug registers.
  - RAS registers.
  - Generic Timer registers.
  - Optionally, Performance Monitor registers.
- Optionally, one or more of the following groups of registers that are defined in other Arm architecture specifications:
  - Trace System registers, as defined in the *Arm® Embedded Trace Macrocell Architecture Specification, ETMv4*.
  - *Generic Interrupt Controller* (GIC) System registers, see The Arm Generic Interrupt Controller System registers.

For information about the AArch64 System registers, see AArch64 System Register Descriptions.

A1.3.3.1 The Arm Generic Interrupt Controller System registers

From version 3 of the Arm Generic Interrupt Controller architecture, GICv3, the GIC architecture specification defines a System register interface to some of its functionality. The System register summaries in this Manual include these registers, see:

- About the GIC System registers, for more information about the AArch64 GIC System registers.

These sections give only short overviews of the GIC System registers. For more information, including descriptions of the registers, see the *ARM® Generic Interrupt Controller Architecture Specification, GIC architecture version 3.0 and version 4.0* (ARM IHI 0069).

Note

The programmers' model for earlier versions of the GIC architecture is wholly memory-mapped.

A1.3.4 Arm Debug

Armv8 and later architectures support the following:

Self-hosted debug

In this model, the PE generates *debug exceptions*. Debug exceptions are part of the Armv8 Exception model.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

External debug

In this model, *debug events* cause the PE to enter *Debug state*. In Debug state, the PE is controlled by an external debugger.

All Armv8 and later implementations support both models. The model chosen by a particular user depends on the debug requirements during different stages of the design and development life cycle of the product. For example, external debug might be used during debugging of the hardware implementation and OS bring-up, and self-hosted debug might be used during application development.

For more information about self-hosted debug:

- In AArch64 state, see AArch64 Self-hosted Debug.

For more information about external debug, see External Debug.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A1.4 Supported data types

The Arm architecture supports the following integer data types:

Byte 8 bits.

Halfword 16 bits.

Word 32 bits.

Doubleword 64 bits.

Quadword 128 bits.

The architecture also supports the following floating-point data types:

- Half-precision. See Half-precision floating-point formats.
- Single-precision. See Single-precision floating-point format.
- Double-precision. See Double-precision floating-point format.

It also supports vectors, where a register holds a vector of multiple elements, each the same size and data type. See Advanced SIMD vector formats

The architecture provides the following:

- A general-purpose register file.
- A SIMD&FP register file.

In each of these, the possible register widths depend on the Execution state.

In AArch64 state:

- The general-purpose register file contains 64-bit registers:
  - Many instructions can access these registers as 64-bit registers, or as 32-bit registers using only the bottom 32 bits.
- The SIMD&FP register file contains 128-bit registers:
  - While the AArch64 vector registers support 128-bit vectors, the effective vector length can be 64-bits or 128-bits depending on the A64 instruction encoding used, see Instruction Mnemonics.

For more information on the register files in AArch64 state, see Registers in AArch64 Execution state.

A1.4.1 Advanced SIMD vector formats

In an implementation that includes the Advanced SIMD instructions that operate on the SIMD&FP register file, a register can hold one or more packed elements, all of the same size and type. In AArch64 state, the explicit combination of a register, number of elements, and element size describes a vector of elements. The vector is considered to be a one-dimensional array of elements of the data type specified in the instruction.

Vector indices are in the range 0 to (number of elements - 1). An index of 0 refers to the least significant bits of the vector.

A1.4.1.1 Advanced SIMD vector formats in AArch64 state

In AArch64 state, the SIMD&FP registers can be referred to as V*n*, where *n* is a value from 0 to 31.

The SIMD&FP registers support the following data formats for loads, stores, and data-processing operations:

- A single, scalar, element in the least significant bits of the register. See Registers in AArch64 Execution state.
- A 128-bit vector of byte, halfword, word, or doubleword elements. See Figure A1-1.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- A 64-bit vector of byte, halfword, word, or doubleword elements. See Figure A1-1.

For vectors, the element sizes are defined in Table A1-1, with the vector format described as:

- For a 128-bit vector: Vn.{2D, 4S, 8H, 16B}.
- For a 64-bit vector: Vn.{1D, 2S, 4H, 8B}.

Table A1-1 SIMD elements in AArch64 state

Mnemonic	Size
B	8 bits
H	16 bits
S	32 bits
D	64 bits

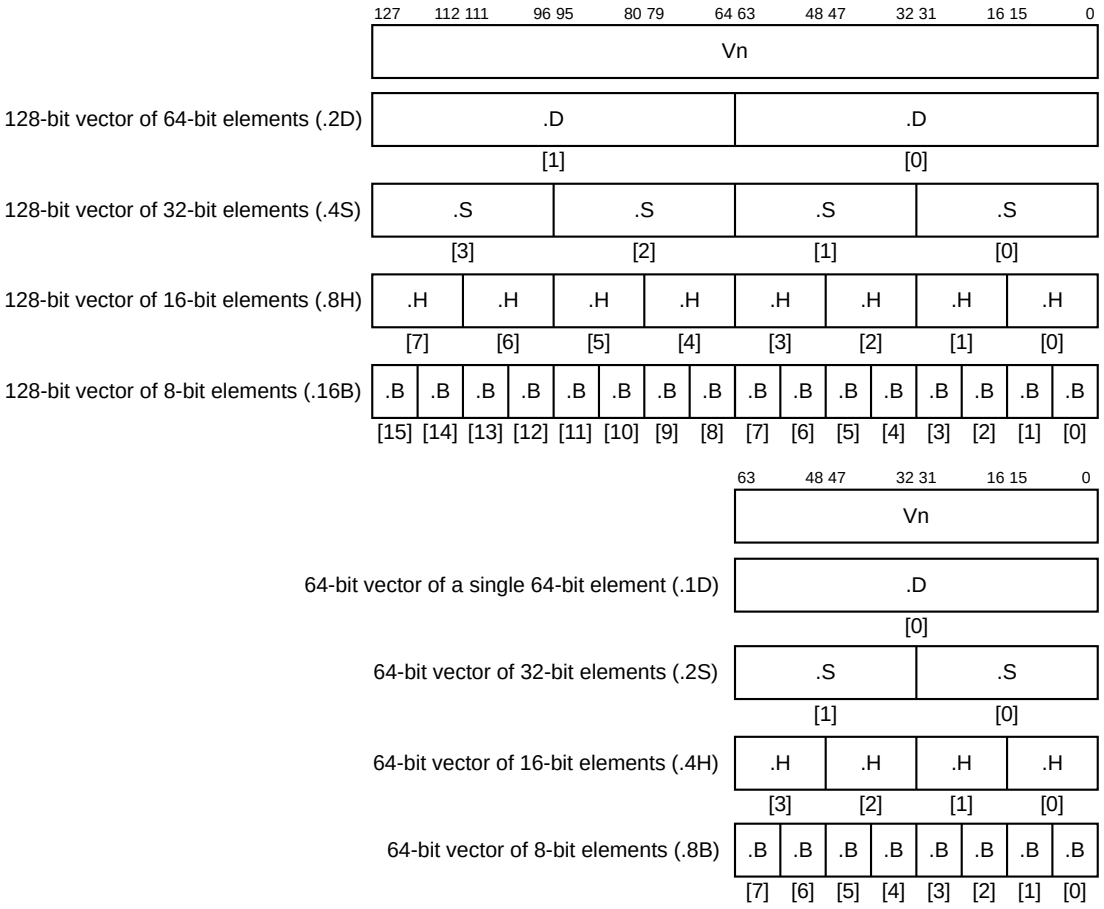


Figure A1-1 SIMD vectors in AArch64 state

A1.4.2 Half-precision floating-point formats

The Arm architecture supports two half-precision floating-point formats:

- IEEE half-precision, as described in the IEEE 754-2008 standard.

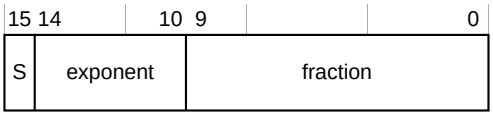
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Arm *alternative half-precision* format.

Both formats can be used for conversions to and from other floating-point formats. FPCR.AHP controls the format in AArch64 state. FEAT_FP16 adds half-precision data-processing instructions, which always use the IEEE format. These instructions ignore the value of the relevant AHP field, and behave as if it has an Effective value of 0.

The description of IEEE half-precision includes Arm-specific details that are left open by the standard, and is only an introduction to the formats and to the values they can contain. For more information, especially on the handling of infinities, NaNs, and signed zeros, see the IEEE 754 standard.

For both half-precision floating-point formats, the layout of the 16-bit format is the same. The format is:



The interpretation of the format depends on the value of the exponent field, bits[14:10] and on which half-precision format is being used.

0 < exponent < `0x1F`

The value is a normalized number and is equal to:

$(-1)^S \times 2^{(exponent-15)} \times (1.fraction)$

The minimum positive normalized number is 2^{-14}, or approximately 6.104×10^{-5}.

The maximum positive normalized number is $(2 - 2^{-10}) \times 2^{15}$, or 65504.

Larger normalized numbers can be expressed using the alternative format when the exponent == `0x1F`.

exponent == 0

The value is either a zero or a denormalized number, depending on the fraction bits:

fraction == 0

The value is a zero. There are two distinct zeros:

+0 when S==0.

-0 when S==1.

fraction != 0

The value is a denormalized number and is equal to:

$(-1)^S \times 2^{-14} \times (0.fraction)$

The minimum positive denormalized number is 2^{-24}, or approximately 5.960×10^{-8}.

Half-precision denormalized numbers are not flushed to zero by default. When FEAT_FP16 is implemented, the FPCR.FZ16 bit controls whether flushing denormalized numbers to zero is enabled for half-precision data-processing instructions. For details, see Flushing denormalized numbers to zero.

exponent == `0x1F`

The value depends on which half-precision format is being used:

IEEE half-precision

The value is either an infinity or a Not a Number (NaN), depending on the fraction bits:

fraction == 0

The value is an infinity. There are two distinct infinities:

+infinity When S==0. This represents all positive numbers that are too big to be represented accurately as a normalized number.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

-infinity When S==1. This represents all negative numbers with an absolute value that is too big to be represented accurately as a normalized number.

fraction != 0

The value is a NaN, and is either a *quiet NaN* or a *signaling NaN*.

The two types of NaN are distinguished by their most significant fraction bit, bit[9]:

bit[9] == 0 The NaN is a signaling NaN. The sign bit can take any value, and the remaining fraction bits can take any value except all zeros.

bit[9] == 1 The NaN is a quiet NaN. The sign bit and remaining fraction bits can take any value.

Alternative half-precision

The value is a normalized number and is equal to:

$-1^{S} \times 2^{16} \times (1.\text{fraction})$

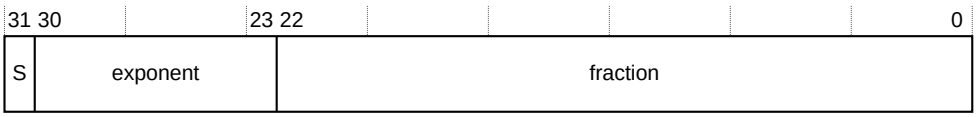
The maximum positive normalized number is $(2-2^{-10}) \times 2^{16}$ or 131008.

A1.4.3 Single-precision floating-point format

The single-precision floating-point format is as defined by the IEEE 754 standard.

This description includes Arm-specific details that are left open by the standard. It is only intended as an introduction to the formats and to the values they can contain. For full details, especially of the handling of infinities, NaNs, and signed zeros, see the IEEE 754 standard.

A single-precision value is a 32-bit word with the format:



The interpretation of the format depends on the value of the exponent field, bits[30:23]:

0 < exponent < `0xFF`

The value is a normalized number and is equal to:

$(-1)^{S} \times 2^{(\text{exponent} - 127)} \times (1.\text{fraction})$

The minimum positive normalized number is 2^{-126}, or approximately 1.175×10^{-38}.

The maximum positive normalized number is $(2 - 2^{-23}) \times 2^{127}$, or approximately 3.403×10^{38}.

exponent == 0

The value is either a zero or a denormalized number, depending on the fraction bits:

fraction == 0

The value is a zero. There are two distinct zeros:

+0 When S==0.

-0 When S==1.

These usually behave identically. In particular, the result is *equal* if +0 and -0 are compared as floating-point numbers. However, they yield different results in some circumstances. For example, the sign of the infinity produced as the result of dividing

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

by zero depends on the sign of the zero. The two zeros can be distinguished from each other by performing an integer comparison of the two words.

fraction != 0

The value is a denormalized number and is equal to:

$(-1)^S \times 2^{-126} \times (0.\text{fraction})$

The minimum positive denormalized number is 2^{-149}, or approximately 1.401×10^{-45}.

Denormalized numbers are optionally flushed to zero in floating-point processing and in Advanced SIMD processing in AArch64 state. For details, see Flushing denormalized numbers to zero.

exponent == `0xFF`

The value is either an *infinity* or a *Not a Number* (NaN), depending on the fraction bits:

fraction == 0

The value is an infinity. There are two distinct infinities:

+infinity When S==0. This represents all positive numbers that are too big to be represented accurately as a normalized number.

-infinity When S==1. This represents all negative numbers with an absolute value that is too big to be represented accurately as a normalized number.

fraction != 0

The value is a NaN, and is either a *quiet NaN* or a *signaling NaN*.

The two types of NaN are distinguished by their most significant fraction bit, bit[22]:

bit[22] == 0

The NaN is a signaling NaN. The sign bit can take any value, and the remaining fraction bits can take any value except all zeros.

bit[22] == 1

The NaN is a quiet NaN. The sign bit and remaining fraction bits can take any value.

For details of the *default NaN*, see Default NaN.

Note

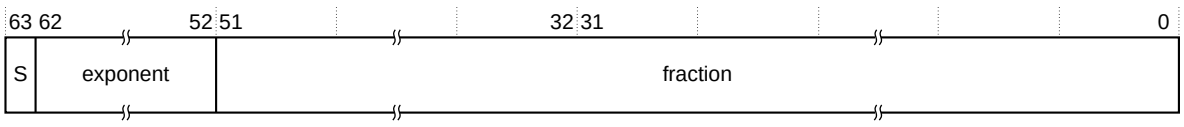
NaNs with different sign or fraction bits are distinct NaNs, but this does not mean software can use floating-point comparison instructions to distinguish them. This is because the IEEE 754 standard specifies that a NaN compares as *unordered* with everything, including itself.

A1.4.4 Double-precision floating-point format

The double-precision floating-point format is as defined by the IEEE 754 standard. Double-precision floating-point is supported by both SIMD and floating-point instructions in AArch64 state.

This description includes IMPLEMENTATION SPECIFIC details that are left open by the standard. It is only intended as an introduction to the formats and to the values they can contain. For full details, especially of the handling of infinities, nabs, and signed zeros, see the IEEE 754 standard.

A double-precision value is a 64-bit doubleword, with the format:



Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Double-precision values represent numbers, infinities, and NaNs in a similar way to single-precision values, with the interpretation of the format depending on the value of the exponent:

0 < exponent < `0x7FF`

The value is a normalized number and is equal to:

$(-1)^S \times 2^{(\text{exponent}-1023)} \times (1.\text{fraction})$

The minimum positive normalized number is 2^{-1022}, or approximately 2.225×10^{-308}.

The maximum positive normalized number is $(2 - 2^{-52}) \times 2^{1023}$, or approximately 1.798×10^{308}.

exponent == 0

The value is either a zero or a denormalized number, depending on the fraction bits:

fraction == 0

The value is a zero. There are two distinct zeros that behave in the same way as the two single-precision zeros:

+0 when S==0.

-0 when S==1.

fraction != 0

The value is a denormalized number and is equal to:

$(-1)^S \times 2^{-1022} \times (0.\text{fraction})$

The minimum positive denormalized number is 2^{-1074}, or approximately 4.941×10^{-324}.

Optionally, denormalized numbers are flushed to zero in floating-point calculations. For details, see Flushing denormalized numbers to zero.

exponent == `0x7FF`

The value is either an infinity or a NaN, depending on the fraction bits:

fraction == 0

The value is an infinity. As for single-precision, there are two infinities:

+infinity When S==0.

-infinity When S==1.

fraction != 0

The value is a NaN, and is either a *quiet NaN* or a *signaling NaN*.

The two types of NaN are distinguished by their most significant fraction bit, bit[51] of the doubleword:

bit[51] == 0

The NaN is a signaling NaN. The sign bit can take any value, and the remaining fraction bits can take any value except all zeros.

bit[51] == 1

The NaN is a quiet NaN. The sign bit and the remaining fraction bits can take any value.

For details of the *default NaN*, see Default NaN.

Note

NaNs with different sign or fraction bits are distinct NaNs, but this does not mean software can use floating-point comparison instructions to distinguish them. This is because the IEEE 754 standard specifies that a NaN compares as *unordered* with everything, including itself.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A1.4.5 Conversion between floating-point and fixed-point values

The Arm architecture supports the conversion of a scalar floating-point value to or from a signed or unsigned fixed-point value in a general-purpose register. Conversion instructions take an argument, #fbits, that specifies the number of fraction bits in the fixed-point number. That is, #fbits indicates that the general-purpose register holds a fixed-point number with fbits bits after the binary point, where fbits is in the range 1 to 64 for a 64-bit general-purpose register, or 1 to 32 for a 32-bit general-purpose register:

- For a 64-bit register X_d:
  - The integer part is $X_d[63:\#fbits]$.
  - The fractional part is $X_d[(\#fbits-1):0]$.
- For a 32-bit register W_d or R_d:
  - The integer part is $W_d[31:\#fbits]$ or $R_d[31:\#fbits]$.
  - The fractional part is $W_d[(\#fbits-1):0]$ or $R_d[(\#fbits-1):0]$.

These instructions can cause the following floating-point exceptions:

Invalid Operation When the floating-point input is NaN or Infinity or when a numerical value cannot be represented within the destination register.

Inexact When the numeric result differs from the input value.

Input Denormal When flushing denormalized numbers to zero is enabled and the denormal input is replaced by a zero, see Flushing denormalized numbers to zero and Input Denormal exceptions.

Note

An out of range fixed-point result is saturated to the destination size.

For more information, see Floating-point exceptions and exception traps.

A1.4.6 Polynomial arithmetic over {0, 1}

Some SIMD instructions that operate on SIMD&FP registers can operate on polynomials over {0, 1}, see Supported data types. The polynomial data type represents a polynomial in x of the form $b_{n-1}x^{n-1} + \ldots + b_1x + b_0$ where b_k is bit[k] of the value.

The coefficients 0 and 1 are manipulated using the rules of Boolean arithmetic:

- $0 + 0 = 1 + 1 = 0$.
- $0 + 1 = 1 + 0 = 1$.
- $0 \times 0 = 0 \times 1 = 1 \times 0 = 0$.
- $1 \times 1 = 1$.

That is:

- Adding two polynomials over {0, 1} is the same as a bitwise exclusive-OR.
- Multiplying two polynomials over {0, 1} is the same as integer multiplication except that partial products are exclusive-ORed instead of being added.

A64 provides instructions for performing polynomial multiplication of 8-bit values, see PMUL and PMULL, PMULL2.

The Cryptographic Extension adds the ability to perform long polynomial multiplies of 64-bit values. See PMULL, PMULL2.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A1.4.6.1 Pseudocode description of polynomial multiplication

In pseudocode, polynomial addition is described by the EOR operation on bitstrings.

Polynomial multiplication is described by the `PolynomialMult()` function defined in R-profile Architecture Pseudocode.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A1.5 Floating-point support

The architecture includes the following types of floating-point instructions:

- Scalar floating-point instructions that operate on the lowest numbered element of the SIMD&FP registers.
- Advanced SIMD floating-point instructions that operate on multiple elements of the SIMD&FP registers.

The architecture can provide the following levels of floating-point support:

- No floating-point support. This option is licensed only for implementations targeting specialized markets.
- Advanced SIMD and floating-point support.

The architecture supports the following floating-point formats:

Table A1-2 Floating-point formats

	AArch64 state
IEEE 754 half-precision	Conversion instructions to and from IEEE 754 half-precision.
	Supported for data processing if FEAT_FP16 is implemented.
Arm *alternative half-precision*	Conversion instructions to and from non-IEEE 754 Arm alternative half-precision.
IEEE 754 single-precision	Supported for scalar and vector data processing.
IEEE 754 double-precision	Supported for scalar and vector data processing.

For many floating-point instructions, there are configurable behaviors, such as:

- Configurable rounding modes. See Rounding.
- Configurable trapping of floating-point exceptions. See Floating-point exceptions and exception traps.
- Configurable non-IEEE 754 *Default NaN* behavior. See NaN handling and the Default NaN.
- Configurable non-IEEE 754 flushing to zero of denormalized numbers. See Flushing denormalized numbers to zero.

If floating-point exception trapping is supported, then unless stated otherwise, when a floating-point exception is not trapped, that exception causes a cumulative status register bit to be set to 1 and the operation produces a default result. For more information, see Floating-point exceptions and exception traps.

In AArch64 state, the Floating-point Control Register, FPCR, controls floating-point operation, and the Floating-point Status Register, FPSR, returns floating-point status information.

In AArch64 state, the PSTATE.{N,Z,C,V} condition flags are updated by floating-point comparison operations.

The remainder of this section contains:

- Instruction support.
- Floating-point standards, and terminology.
- Arm standard floating-point input and output values.
- Flushing denormalized numbers to zero.
- NaN handling and the Default NaN
- Rounding

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Floating-point exceptions and exception traps

A1.5.1 Instruction support

The floating-point instructions support:

- Load and store for single elements and vectors of multiple elements.

  Note
  Single elements are also referred to as scalar elements.

- Data processing on single and multiple elements.
- Complex number arithmetic.
- Floating-point conversion between different levels of precision.
- Conversion between floating-point and integer types.
- Floating-point rounding.
- Atomic floating-point in-memory operations.

For more information on floating-point instructions in AArch64 state, see A64 Instruction Set Overview.

A1.5.2 Floating-point standards, and terminology

The Arm architecture includes support for all the required features of ANSI/IEEE Std 754-2008, *IEEE Standard for Binary Floating-Point Arithmetic*, referred to as IEEE 754-2008. However, some terms in this Manual are based on the 1985 version of this standard, referred to as IEEE 754-1985:

- Arm floating-point terminology generally uses the IEEE 754-1985 terms. This section summarizes how IEEE 754-2008 changes these terms.
- References to IEEE 754 that do not include the issue year apply to either issue of the standard.

Table A1-3 shows how the terminology in this Manual differs from that used in IEEE 754-2008.

Table A1-3 Floating-point terminology

This manual	IEEE 754-2008
Normalized[a]	Normal
Denormal, or denormalized	Subnormal
Round towards Minus Infinity (RM)	roundTowardsNegative
Round towards Plus Infinity (RP)	roundTowardsPositive
Round towards Zero (RZ)	roundTowardZero
Round to Nearest (RN)	roundTiesToEven
Round to Nearest with Ties to Away	roundTiesToAway
Rounding mode	Rounding-direction attribute

a. *Normalized number* is used in preference to *normal number*, because of the other specific uses of *normal* in this Manual.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A1.5.3 Arm standard floating-point input and output values

The Arm architecture provides full IEEE 754 floating-point arithmetic support.

Arm standard floating-point arithmetic supports the following input formats defined by the IEEE 754 floating-point standard:

- Zeros.
- Normalized numbers.
- Denormalized numbers are flushed to 0 before floating-point operations, see Flushing denormalized numbers to zero.
- NaNs.
- Infinities.

Arm standard floating-point arithmetic supports the Round to Nearest (roundTiesToEven) rounding mode defined by the IEEE 754 standard.

Arm standard floating-point arithmetic supports the following output result formats defined by the IEEE 754 standard:

- Zeros.
- Normalized numbers.
- Results with an absolute value less than the minimum normalized number are flushed to zero, see Flushing denormalized numbers to zero.
- NaNs produced in floating-point operations are always the default NaN, see Default NaN.
- Infinities.

Note

AArch64 Advanced SIMD floating-point arithmetic is performed using the rounding mode selected by the FPCR.

A1.5.4 Flushing denormalized numbers to zero

Calculations involving denormalized numbers and Underflow exceptions can reduce the performance of floating-point processing. For many algorithms, replacing the denormalized operands and intermediate results with zeros can recover this performance, without significantly affecting the accuracy of the final result. Arm floating-point implementations allow denormalized numbers to be flushed to zero to permit this optimization.

If a number `value` satisfies the condition `0 < Abs(value) < MinNorm`, it is treated as a denormalized number.

`MinNorm` is defined as follows:

- For OFP8 E4M3 numbers, `MinNorm` is 2^{-6}.
- For half-precision and OFP8 E5M2 numbers, `MinNorm` is 2^{-14}.
- For single-precision and BFloat16 numbers, `MinNorm` is 2^{-126}.
- For double-precision numbers, `MinNorm` is 2^{-1022}.

Flushing denormals to zero is incompatible with the IEEE 754 standard, and must not be used when IEEE 754 compatibility is a requirement. Enabling flushing of denormals to zero must be done with care. Although it can improve performance on some algorithms, there are significant limitations on its use. These are application-dependent:

- On many algorithms, it has no noticeable effect, because the algorithm does not usually process denormalized numbers.
- On other algorithms, it can cause exceptions to occur and can seriously reduce the accuracy of the results of the algorithm.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A1.5.4.1 Flushing denormalized inputs to zero

If flushing denormalized inputs to zero is enabled for an instruction and a data type, and an input to that instruction is a denormalized number of that data type, the input operand is flushed to zero, and its sign bit is not changed.

If a floating-point operation has a denormalized input that is flushed to zero, for all purposes within the instruction other than calculating Input Denormal floating-point exceptions, all inputs that are denormalized numbers are treated as though they were zero with the same sign as the input.

For floating-point instructions, if the instruction processes half-precision inputs, flushing denormalized inputs to zero can be controlled as follows:

- If FPCR.FZ16 == 0, denormalized half-precision inputs are not flushed to zero.
- If FPCR.FZ16 == 1:
  - Denormalized half-precision inputs to data processing instructions are flushed to zero.
  - Denormalized half-precision inputs to convert instructions are not flushed to zero.

If FPCR.FZ == 0, for all inputs other than half-precision, FPCR.FZ does not cause denormalized inputs to be flushed to zero, although other factors might cause denormalized inputs to be flushed to zero.

A1.5.4.2 Flushing denormalized outputs to zero

If a denormalized output is flushed to zero, the output is returned as zero with the same sign bit as the denormalized output value.

If FPCR.FZ == 0, then unless otherwise specified, for all outputs other than half-precision, FPCR.FZ does not cause denormalized outputs to be flushed to zero, although other factors might cause denormalized outputs to be flushed to zero.

If FPCR.FZ == 1, for all outputs other than half-precision, for floating-point instructions other than `FABS`, `FNEG`, `FMAX`, `FMAXP`, `FMAXV`, `FMIN`, `FMINP`, `FMINV`, `BFMIN`, and `BFMAX`, denormalized outputs are flushed to zero.

If FPCR.FZ16 == 0, denormalized half-precision outputs are not flushed to zero.

If FPCR.FZ16 == 1, for floating-point instructions other than `FABS`, `FNEG`, `FMAX`, `FMAXP`, `FMAXV`, `FMIN`, `FMINP`, `FMINV`, `BFMIN`, and `BFMAX`, denormalized half-precision outputs are flushed to zero.

A1.5.5 NaN handling and the Default NaN

The IEEE 754 standard defines a NaN as a number with all exponent bits set to 1 and a nonzero number in the mantissa, and specifies that the sign bit of a NaN has no significance.

The Arm architecture additionally defines the following in Table A1-4:

- A *Default NaN*, compliant with the IEEE 754 standard.
- A BFloat16 NaN, that follows the encoding in BFloat16 floating-point format.
- OFP8 NaN, that follow the encodings in FP8 floating-point formats.

For a quiet NaN output derived from a signaling NaN operand, the most significant fraction bit is set to 1.

A PE is forbidden to generate a NaN whose value is strongly correlated to the values of non-NaN inputs as a speculative result of a floating-point calculation not involving NaN inputs.

A1.5.5.1 Default NaN

A Default NaN is encoded as described in Table A1-4.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table A1-4 Default NaN encodings

	Half-precision, IEEE Format	Single-precision	Double-precision	BFloat16	OFP8 E4M3	OFP8 E5M2
Exponent	`0x1F`	`0xFF`	`0x7FF`	`0xFF`	`0xF`	`0x1F`
Fraction	Bit[9] == '1', bits[8:0] == 0	Bit[22] == '1', bits[21:0] == 0	Bit[51] == '1', bits[50:0] == 0	Bit[6] == '1', bits[5:0] == 0	Bits[2:0] == '111'	Bit[1] == '1', bit[0] == '0'

When FPCR.DN == 1, if any input to a floating-point operation is a NaN, the output is a Default NaN.

Floating-point instructions that detect an Invalid Operation condition for a reason other than one of its inputs being a NaN, produce a Default NaN as the result.

A1.5.5.2 NaN handling

The IEEE 754 standard does not specify which input NaN is used as the output NaN. Therefore, where the Arm architecture specifies which input NaN to use, this is an addition to the requirements in the IEEE 754 standard.

Depending on the operation, the exact value of a derived quiet NaN output might have both a different sign and different fraction bits from its source. See individual instruction descriptions for details.

A1.5.5.3 NaN propagation

For NaN propagation rules for:

- BFDOT and BFMMLA, see BFloat16 behaviors for instructions that compute sum-of-products.
- FP8 instructions, see Summary of FP8 instruction behaviors.

Otherwise, behavior is as described in this section.

If an output NaN is derived from one of the operands, how the input NaN propagates to the output depends on the instruction and the number of operands.

If an output NaN is derived from an input NaN and if the size of the output format is the same as the input format, then all of the following apply:

- If the input NaN is a quiet NaN, the output NaN is derived from the input NaN.
- If the input NaN is a signaling NaN, the output NaN is derived as follows:
  - If the handling of a signaling NaN by the instruction detects an Invalid Operation condition, the output NaN is derived from a quietened version of the input NaN.
  - If the handling of a signaling NaN by the instruction does not detect an Invalid Operation condition, the output NaN is derived from the input NaN.

If an output NaN is derived from an input NaN and if the size of the output format is larger than the input format, all of the following apply:

- If the input NaN is a quiet NaN, the output NaN is the same as the input NaN except that the mantissa is zero-extended in the low-order bit to fit the output format, and the exponent field is set to all ones.
- If the input NaN is a signaling NaN, the output NaN is the quieted version of the input NaN, except that the mantissa is zero-extended in the low-order bits and the exponent field is set to all ones.

If an output NaN is derived from an input NaN and if the size of the output format is smaller than the input format, all of the following apply:

- If the input NaN is a quiet NaN, the output NaN is the same as the input NaN except that the mantissa is truncated in the lower-order bits to fit the output format, and the exponent field is set to all ones.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the input NaN is a signaling NaN, the output NaN is the quieted version of the input NaN except that the mantissa is truncated in the lower-order bits to fit the output format, and the exponent field is set to all ones.

For the following descriptions, the term “first operand” and “second operand” relate to the left-to-right ordering of the arguments of the pseudocode function that describes the operation.

A1.5.6 Rounding

The rounding mode specifies how the exact result of a floating-point operation is rounded to a value in the destination format.

The rounding mode is either determined by the rounding mode control field FPCR.RMode or by the instruction.

FPCR.RMode can select one of:

- Round to Nearest (RN) mode.
- Round towards Plus Infinity (RP) mode.
- Round towards Minus Infinity (RM) mode.
- Round towards Zero (RZ) mode.

The following additional rounding behaviors are not selected by FPCR.RMode, but are used by some instructions:

- Round to Odd.
- Round to Nearest with Ties to Away.

A1.5.6.1 Round to Nearest (RN) mode

Round to Nearest rounding mode rounds the exact result of a floating-point operation to a value that is representable in the destination format as follows:

- If the value before rounding has an absolute value that is too large to represent in the output format, the rounded value is an Infinity. The sign of the rounded value is the same as the sign of the value before rounding.
- If the value before rounding has an absolute value that is not too large to represent in the output format, the result is calculated as follows:
  - If the two nearest floating-point numbers bracketing the value before rounding are equally near, the result is the number with an even least significant digit.
  - If the two nearest floating-point numbers bracketing the value before rounding are not equally near, the result is the floating-point number nearest to the value before rounding.

A1.5.6.2 Round towards Plus Infinity (RP) mode

Round towards Plus Infinity rounding mode rounds the exact result of a floating-point operation to a value that is representable in the destination format. The result is the floating-point number in the output format that is closest to and not less than the value before rounding. The result can be plus infinity.

A1.5.6.3 Round towards Minus Infinity (RM) mode

Round towards Minus Infinity rounding mode rounds the exact result of a floating-point operation to a value that is representable in the destination format. The result is the number in the output format that is closest to and not greater than the value before rounding. The result can be minus infinity.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A1.5.6.4 Round towards Zero (RZ) mode

Round towards Zero rounding mode rounds the exact result of a floating-point operation to a value that is representable in the destination format. The result is the floating-point number in the output format that is closest to and not greater in absolute value than the value before rounding.

A1.5.6.5 Round to Nearest with Ties to Away

Round to Nearest with Ties to Away rounds the exact result of a floating-point operation to a value that is representable in the destination format. The result is calculated as follows:

- If the two nearest floating-point numbers bracketing the value before rounding are equally near, the result is the number with the largest absolute value.
- If the two nearest floating-point numbers bracketing the value before rounding are not equally near, the result is the floating-point number nearest to the value before rounding.

Round to Nearest with Ties to Away rounding is used by FCVTAS (scalar), FCVTAS (vector), FCVTAU (scalar), FCVTAU (vector), FRINTA (scalar), and FRINTA (vector).

A1.5.6.6 Round to Odd

Round to Odd is not defined by IEEE 754.

Round to Odd rounds the exact result of a floating-point operation to a value that is representable in the destination format. If the result of the rounding is inexact, the least significant bit of the mantissa is forced to 1.

FCVTXN, FCVTXN2 for which Round to Odd rounding can avoid double rounding errors when a floating-point value is converted to a lower precision destination format through an intermediate precision format.

Example A1-1 Converting 64-bit floating-point format to 16-bit floating-point format

A 64-bit floating-point value can be converted to a correctly rounded 16-bit floating-point value using the following steps:

1. Use an `FCVTXN` instruction to produce a 32-bit value.
2. Use another instruction with the required rounding mode to convert the 32-bit value to the final 16-bit floating-point value.

A1.5.7 Floating-point exceptions and exception traps

Execution of a floating-point instruction, or execution of an Advanced SIMD instruction that performs floating-point operations, can generate an exceptional condition, called a *floating-point exception*.

Note

In AArch64 state, an Advanced SIMD instruction that operates on floating-point values can perform multiple floating-point operations. Therefore, this section describes the handling of a floating-point exception on an *operation*, rather than on an *instruction*.

The architecture does not support asynchronous reporting of floating-point exceptions.

For each of the following floating-point exceptions, it is IMPLEMENTATION DEFINED whether an implementation includes synchronous exception generation:

- Input Denormal.
- Inexact.
- Underflow.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Overflow.
- Divide by Zero.
- Invalid Operation.

If an implementation does not support synchronous exception generation from a floating-point exception, then that synchronous exception is never generated and all statements about synchronous exception generation from that floating-point exception do not apply to the implementation.

Synchronous exception generation by floating-point exceptions is enabled using the FPCR as follows:

- For each floating-point exception that supports synchronous exception generation, the relevant control bit chosen from FPCR.{IDE, IXE, UFE, OFE, DZE, IOE} is used to enable synchronous exception generation.
- For each floating-point exception that does not support synchronous exception generation, the relevant bit chosen from FPCR.{IDE, IXE, UFE, OFE, DZE, IOE} is RAZ/WI.

For information about FP8 instruction behaviors, see Summary of FP8 instruction behaviors.

A1.5.7.1 Inexact exceptions

The cumulative floating-point exception bit FPSR.IXC and the trap enable bit FPCR.IXE both relate to Inexact exceptions.

If a result is not flushed to zero, and the result does not equal the result computed with unbounded exponent range and unbounded precision, then an Inexact exception is generated.

A1.5.7.2 Underflow exceptions

The cumulative floating-point exception bit FPSR.UFC, and the trap enable bit FPCR.UFE both relate to Underflow exceptions.

If the result of a floating-point operation is a denormalized number that is not flushed to zero, then:

- If the result is exact, an Underflow floating-point exception is generated only if FPCR.UFE is 1.
- If the result is inexact, an Underflow floating-point exception is always generated.

If the result of a floating-point operation is a denormalized number that is flushed to zero, then the Underflow floating-point exception is generated, however this floating-point exception is not trapped regardless of the value of FPCR.UFE.

A1.5.7.3 Overflow exceptions

The cumulative floating-point exception bit FPSR.OFC, and the trap enable bit FPCR.OFE both relate to Overflow exceptions.

If the output of an instruction rounded with an unbounded exponent is greater than the maximum normalized number for the output precision, an Overflow exception is generated.

If an untrapped Overflow exception is generated, the result is determined by the selected rounding behavior and the sign of the result before rounding, as follows:

- Round to Nearest carries all overflows to infinity with the sign of the result before rounding.
- Round towards Plus Infinity carries negative overflows to the most negative finite number of the output precision, and carries positive overflows to plus infinity.
- Round towards Minus Infinity carries positive overflows to the largest finite number of the output precision, and carries negative overflows to minus infinity.
- Round towards Zero carries all overflows to the output precision's largest finite number with the sign of the result before rounding.
- Round to Nearest Ties to Away carries all overflows to infinity with the sign of the result before rounding.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Round to Odd, when used by convert instructions, carries all overflows to the output precision's largest finite number with the sign of the result before rounding.
- Round to Odd, when used by some BFloat16 instructions, carries all overflows to infinity with the sign of the result before rounding.

A1.5.7.4 Divide by Zero exceptions

The cumulative floating-point exception bit FPSR.DZC, and the trap enable bit FPCR.DZE both relate to Divide by Zero exceptions.

If a floating-point operation divides a finite nonzero number by zero, a Divide by Zero exception is generated.

If a floating-point operation divides a finite nonzero number by zero, and the Divide by Zero exception is untrapped, the result is a correctly signed infinity.

A1.5.7.5 Invalid Operation exceptions

The cumulative floating-point exception bit FPSR.IOC, and the trap enable bit FPCR.IOE both relate to Invalid Operation exceptions.

For any floating-point instruction that performs a floating-point operation, if any of the following apply, the instruction generates an Invalid Operation exception:

- At least one operand is a signaling NaN.
- Subtracting two equal-magnitude infinities that are of the same sign.
- Multiplying a zero by an infinity.
- Dividing a zero by a zero.
- Dividing an infinity by an infinity.
- Taking the square root of an operand that is less than zero.

If the input is one of: a quiet NaN, an infinity, or a number that overflows the values that can be represented in the output format, and if another exception is not generated to signal the condition, then a conversion from floating-point to either integer or fixed-point format, generates an Invalid Operation exception.

For the signaling compare instructions FCMPE, if either of the source operands is any type of NaN, the instruction generates an Invalid Operation floating-point exception.

A1.5.7.6 Handling floating-point exceptions

If an implementation supports synchronous exception generation for floating-point exceptions, the synchronous exceptions generated by the floating-point exception traps are taken to the lowest Exception level that can handle such an exception and that is not at a lower Exception level than where the exception was generated.

If an implementation supports synchronous exception generation for floating-point exceptions in AArch64 state, all of the following apply:

- The registers that are presented to the exception handler are consistent with the state of the PE immediately before the instruction that caused the exception, except that an implementation is permitted to not restore the cumulative floating-point exception bits in the event of such an exception.
- When the execution of separate operations in separate SIMD elements causes multiple floating-point exceptions, the ESR_ELx reports one exception associated with one element that the instruction uses. The architecture does not specify which element is reported.

The `AArch64.FPTrappedException()` and `FPProcessException()` pseudocode functions describe the handling of trapped floating-point exceptions generated in AArch64 state.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A1.5.7.7 Combinations of floating-point exceptions

More than one floating-point exception might occur on the same operation. The architecture permits only the following combinations of floating-point exceptions:

- Overflow with Inexact.
- Underflow with Inexact.

If two floating-point exceptions occur on the same operation, the Input Denormal exception is treated as highest priority and the Inexact exception is treated as lowest priority.

Some floating-point instructions specify more than one floating-point operation, this is indicated by the pseudocode descriptions of the instruction. In these cases, it is possible for one instruction to generate multiple exceptions. Multiple exceptions from one instruction are prioritized as follows:

- If an exception generating operation outputs a result that is used by a second exception generating operation, the exception generated by the operation that outputs the result is treated as higher priority than the exception generated by the second operation that uses the result.
- If exception generating operations do not use the outputs of other exception generating operations, it is CONSTRAINED UNPREDICTABLE which floating-point exception is treated as higher priority. The exception prioritized might differ between different instances of the same two floating-point exceptions being generated on the same operation during execution of the instruction.
- A trapped Underflow exception has priority over a trapped Inexact exception.

If none of the floating-point exceptions caused by an operation is trapped, any floating-point exception that occurs causes the associated cumulative exception flag bit in the FPSR to be set to 1.

When FPCR.AH is 1, if the result of a floating-point operation is a denormalized number that is flushed to zero, both an untrapped Underflow floating-point exception and an Inexact floating-point exception are generated, which always sets the cumulative Underflow exception flag FPSR.UFC to 1 regardless of whether the Inexact floating-point exception is trapped. Otherwise, when a trapped floating-point exception is taken, it is IMPLEMENTATION DEFINED whether the FPSR is restored to the value it had immediately before the instruction that generated the trapped floating-point exception. If the IMPLEMENTATION DEFINED choice is that the FPSR is not restored, it is CONSTRAINED UNPREDICTABLE which trapped floating-point exceptions, if any, are indicated by the corresponding FPSR cumulative exception flag bits being set to 1.

When a trapped floating-point exception is taken, then in the ESR_ELx, all of the following apply:

- The highest priority trapped floating-point exception has a floating-point exception trapped bit set to 1.
- If any other untrapped floating-point exceptions are generated by the same operation, each untrapped exception has a floating-point exception trapped bit set to 0. This applies to both higher priority and lower priority untrapped floating-point exceptions.
- If any lower priority trapped floating-point exceptions are generated by the same operation, for each exception, it is CONSTRAINED UNPREDICTABLE whether the floating-point exception trapped bit is set to 1.

The architectural requirements for floating-point exception prioritization apply only to multiple floating-point exceptions generated on the same element of an Advanced SIMD operation. For trapped floating-point exceptions from Advanced SIMD the architecture does not define the floating-point exception prioritization between different elements of the instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A1.6 The Arm memory model

The Arm memory model supports:

- Generating an exception on an unaligned memory access.
- Restricting access by applications to specified areas of memory.
- Translating *virtual addresses* (VAs) provided by executing instructions to *physical addresses* (PAs).
- Altering the interpretation of multi-byte data between big-endian and little-endian.
- Controlling the order of accesses to memory.
- Controlling caches and address translation structures.
- Synchronizing access to shared memory by multiple PEs.
- Barriers that control and prevent speculative access to memory.

AArch64 state VA supports 64-bit virtual addressing, with the Translation Control Register determining the supported VA range. Execution at EL1 and EL0 supports two independent VA ranges, each with its own translation controls.

Regardless of the Execution state, the *Virtual Memory System Architecture* (VMSA) can translate VAs to blocks or pages of memory anywhere within the supported PA space.

For more information, see:

For execution in AArch64 state

- The AArch64 Application Level Memory Model.
- The AArch64 System Level Memory Model.
- The AArch64 Virtual Memory System Architecture.
- The AArch64 Protected Memory System Architecture

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A1.7 Reliability, Availability, and Serviceability

A1.7.1 Introduction

I_{LMHPD} RAS are three aspects of the *dependability* of a system:

- Reliability, that is, the continuity of correct service.
- Availability, that is, the readiness for correct service.
- Serviceability, that is, the ability to undergo modifications and repairs.

I_{HWHJM} RAS techniques reduce unplanned outages because:

- Transient errors can be detected and corrected before they cause application or system failure.
- Failing components can be identified and replaced.
- Failure can be predicted ahead-of-time to allow replacement during planned maintenance.

A1.7.2 Faults, errors, and failures

R_{NVNNC} Correct service is delivered when the service implements the system function.

I_{QWSVK} Correct service might include:

- Producing correct results.
- Producing results within the time allotted to the task.
- Not divulging secret or secure information.

R_{KRTSC} For the purpose of describing the RAS Extension and RAS System Architecture, deviation from correct service is defined using the following terms:

- A failure is the event of deviation from correct service. This includes data corruption, data loss, and service loss.
- An error is the deviation from correct service. An incorrect value that has an error is corrupt.
- A fault is the cause of the error.

R_{JNBDX} Errors that are present but not detected are latent errors or undetected errors.

I_{TNQPK} In a system with no error detection, all errors are latent errors and are silently propagated by components until they are either masked or cause failure.

I_{GRYKV} The severity of a failure can range from minor to catastrophic:

- The harmful consequences of a minor failure are of a similar cost to the benefits provided by correct service delivery.
- The harmful consequences of a catastrophic failure are orders of magnitude, or even incommensurably, higher than the benefit provided by correct service delivery.

I_{NMGPQ} There are many sources of faults in a system, including both software and hardware faults:

- *Hardware faults* originate in, or affect, hardware.
- *Software faults* affect software, that is programs or data.

The RAS Extension and RAS System Architecture primarily address errors produced from hardware faults. These fall into two main areas:

- Transient faults.
- Non-transient or *persistent faults*.

A1.7.3 General taxonomy of errors

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A1.7.3.1 Error detection

R_{FHXWP} When a component accesses memory or other state, an error might be detected in that memory or state.

I_{WKPVR} The error might be corrected or deferred by the component, or signaled to another component as either a Deferred error or a Detected error.

A1.7.3.2 Error propagation

R_{LRZDN} A transaction occurs when a producer of the transaction passes a value or other signal to a consumer of the transaction.

I_{VYCCX} Transactions are part of the service provided by the producer for the consumer.

I_{RHGDL} In many protocols and service interface definitions, a high-level transaction consists of a sequence of operations, for instance between a *Requester* and a *Completer*.

For the purposes of this manual, the most basic form of a unidirectional transfer between a *producer* and *consumer* is considered as a transaction.

That is, each one of the sequence of operations is considered a separate transaction. For some operations, such as a request, the Requester is producer and the Completer is the consumer. For other operations, such as a response, the Completer is producer and the Requester is the consumer.

R_{SKZZG} An error is propagated by the producer of a transaction when the service interface is incorrect because of the error. The error is propagated to the consumer.

R_{CHCCV} An error is propagated by deviations from correct service, including when any of the following occurs that would not have been permitted to occur had the fault not been activated:

- A corrupt value is passed from producer to consumer.
- A transaction or other operation occurs that should not have occurred.
- A transaction or other operation that should have occurred does not occur.
- A loss of uniprocessor semantics or any other loss of coherency in a multiprocessor coherent system is observed.
- Changing the timing and/or order of transactions or other operations such that the timing and/or order of those transactions or operations is incorrect. In this case, the service interface defines acceptable timings and/or orders for transactions and other operations.

R_{YFBRV} The service interface for a transaction might include means to signal that the transaction is propagating either of the following:

- A Detected error.
- A Deferred error.

R_{BHWVX} An error is silently propagated by the producer of a transaction if the consumer of the transaction cannot detect the error and consumes an undetected error because of the transaction. This might be because of one of the following:

- The error is present on the transaction, but was not detected by the producer. The error is silently propagated by the producer.
- The error is present on the transaction, but was not signaled to the consumer as an error. For example, a corrupt value was passed in the transaction with no indication that it was corrupt. The error is silently propagated by the producer.

R_{FPBYS} A latent, possibly detectable, error is silently propagated by the consumer of an otherwise correct transaction if the transaction causes the error to become undetectable.

Example A1-2

A partial write to a protection granule removes poison, leaving the unchanged portion of the location corrupt. To implement a partial write, the consumer logically reads the current value of the location, modifies the value, and then writes the modified value back. These are internal transactions in the consumer that silently propagate the error. In this example there was no error at the producer nor on the transaction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{QXRLB} Errors might be propagated by components in a system until one of the following occurs:

- They are masked and do not affect the outcome of the system.
  The error might be masked because a corrupt value is discarded or overwritten, or the error is detected and removed.
- They affect the service interface of the system and possibly cause failure. If the error has been silently propagated to the service interface, then:
  - This is a Silent Data Corruption, SDC.
  - The rate of such failures, measured as the number of failures per billion device-hours of operation, is called the Silent Data Corruption Failure-in-Time rate, SDC FIT rate.

  Alternatively, the error might have been detected, causing the system to invoke error handling and recovery. For more information, see *Arm® Reliability, Availability, and Serviceability (RAS) System Architecture, for A-profile architecture* (ARM IHI 0100).

A1.7.3.3 Infected and poisoned

R_{KNHWB} The state of a component becomes infected when the component consumes an Uncorrected error that updates the state.

R_{TZBSW} A value is poisoned in the state of a component if it is marked as being in error, such that a subsequent access of the state will detect the value is so marked and is treated as a Detected error.

I_{YBMFK} Poison is used to defer an error.

A1.7.3.4 Containable and uncontainable

R_{DXQRD} An undetected error is uncontained at the component that failed to detect it.

R_{RJYRQ} A silently propagated error is uncontained at the component that silently propagated it.

R_{GJQNR} A Detected Uncorrected error is uncontainable at the component if it might be uncontained at the component. A Detected Uncorrected error is containable at the component if it is not uncontainable at the component. If the component cannot determine whether a Detected Uncorrected error is uncontainable at the component or containable at the component, then the component treats the Detected Uncorrected error as uncontainable at the component.

I_{MRDMR} An error that is uncontainable at the component might be containable at the system level.

I_{NWZGB} Reporting an error as containable allows software to contain the error. This does not mean that hardware has contained the error.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter A2
R-profile Architecture Extensions

This chapter introduces the Arm R-profile architecture extensions. It contains the following sections:

- About the R-profile architecture extensions.
- Armv8-R architecture extensions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A2.1 About the Armv8-R AArch64 architecture extensions

An architecture extension adds a set of features that are when implemented can make an implementation compliant with the version of the architecture.

Each feature description includes:

- A feature name.
- A brief description of the feature.
- The Execution state the feature is supported in.
- Whether the implementation of the feature is mandatory or OPTIONAL.
- Dependencies on the implementation of other features if any.
- The register field that identifies the presence of the feature.

A2.1.1 Permitted implementation of subsets of the Armv8-R AArch64 architectural features

When a feature is introduced, the architecture specifies when the feature can be implemented. Typically, a feature introduced as part of Armv8.*x* is permitted to be built in Armv8.(*x*-1), subject only to those constraints that require that certain features be implemented together.

An implementation of the Armv8-R AArch64 architecture is based on the Armv8.4 architecture extensions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A2.2 Armv8-R AArch64 architecture extensions

A2.2.1 The Armv8.0 architecture extension

The original Armv8-R architecture is called Armv8.0. It contains mandatory and optional architectural features. Some features must be implemented together. An implementation is Armv8.0 compliant if it includes all of the Armv8.0 architectural features that are mandatory.

An Armv8.0 compliant implementation can additionally include:

- Armv8.0 features that are optional.
- Any arbitrary subset of the architectural features of Armv8.1, subject only to those constraints that require that certain features be implemented together.

FEAT_AA64, PE uses AArch64 after last reboot

This feature is mandatory.

FEAT_AA64EL0, Support for AArch64 at EL0

This feature is mandatory.

FEAT_AA64EL1, Support for AArch64 at EL1

This feature is mandatory.

FEAT_AA64EL2, Support for AArch64 at EL2

This feature is mandatory.

FEAT_AES, Advanced SIMD AES instructions

FEAT_AES provides the AES* instructions to support AES encryption and decryption, AESD, AESE, AESMC, and AESIMC.

FEAT_AES is OPTIONAL from Armv8.0.

If FEAT_AES is implemented, then FEAT_Crypto is implemented.

The following field identifies the presence of FEAT_AES:

- ID_AA64ISAR0_EL1.AES.

FEAT_ASID16, 16 bit ASID

FEAT_ASID16 is OPTIONAL from Armv8.0.

The following field identifies the presence of FEAT_ASID16:

- ID_AA64MMFR0_EL1.ASIDBits.

FEAT_AdvSIMD, Advanced SIMD Extension

FEAT_AdvSIMD includes support for the SISD and SIMD operations. All Armv8-A systems that support standard operating systems with rich application environments also provide hardware support for Advanced SIMD instructions.

FEAT_AdvSIMD is OPTIONAL from Armv8.0.

If FEAT_AdvSIMD is implemented, then FEAT_FP is implemented.

The following fields identify the presence of FEAT_AdvSIMD:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- ID_AA64PFR0_EL1.AdvSIMD.
- EDPFR.AdvSIMD.

For more information, see:

- Supported data types.
- A64 Advanced SIMD and Floating-point Instruction Descriptions.

FEAT_BigEnd, Support for big-endian at EL1 and above

FEAT_BigEnd is OPTIONAL from Armv8.0.

If FEAT_BigEnd is implemented, then FEAT_BigEndEL0 is implemented.

If FEAT_MixedEnd is implemented, then FEAT_BigEnd is implemented.

FEAT_BigEndEL0, Support for big-endian at EL0

FEAT_BigEndEL0 is OPTIONAL from Armv8.0.

If FEAT_MixedEndEL0 is implemented, then FEAT_BigEndEL0 is implemented.

FEAT_CRC32, CRC32 instructions

FEAT_CRC32 introduces the support for the CRC32* instructions that perform cyclic redundancy check calculations.

FEAT_CRC32 is mandatory from Armv8.0.

The following field identifies the presence of FEAT_CRC32:

- ID_AA64ISAR0_EL1.CRC32.

FEAT_CSV2_1p1, Cache Speculation Variant 2

For FEAT_CSV2_1p1, within a hardware-described context, branch targets trained for branches situated at one address can control speculative execution of branches situated at different addresses only in a hard-to-determine way.

FEAT_CSV2_1p1 does not support the SCXTNUM_ELx registers.

FEAT_CSV2_1p1 is OPTIONAL from Armv8.0.

If FEAT_CSV2_1p1 is implemented, then FEAT_CSV2 is implemented.

The following fields identify the presence of FEAT_CSV2_1p1:

- ID_AA64PFR1_EL1.CSV2_frac.
- ID_AA64PFR0_EL1.CSV2.

For more information, see:

- For AArch64, ‘Restrictions on the effects of speculation’.

FEAT_CSV2_1p2, Cache Speculation Variant 2 version 1.2

For FEAT_CSV2_1p2, within a hardware-described context, branch targets trained for branches situated at one address can control speculative execution of branches situated at different addresses only in a hard-to-determine way.

FEAT_CSV2_1p2 introduces the SCXTNUM_ELx registers, but the contexts do not include the SCXTNUM_ELx register contexts.

FEAT_CSV2_1p2 is OPTIONAL from Armv8.0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If FEAT_CSV2_1p2 is implemented, then FEAT_CSV2_1p1 is implemented.

The following fields identify the presence of FEAT_CSV2_1p2:

- ID_AA64PFR1_EL1.CSV2_frac.
- ID_AA64PFR0_EL1.CSV2.

For more information, see:

- For AArch64, ‘Restrictions on the effects of speculation’.

FEAT_CSV2_2, Cache Speculation Variant 2 version 2

FEAT_CSV2_2 introduces the SCXTNUM_ELx registers, which provide a number that can be used to separate out different context numbers within their respective Exception levels for the purpose of protecting against speculative attacks through prediction resources.

FEAT_CSV2_2 is OPTIONAL from Armv8.0.

If FEAT_CSV2_2 is implemented, then FEAT_CSV2 is implemented.

If FEAT_CSV2_2 is implemented, then FEAT_CSV2_1p1 is not implemented.

The following field identifies the presence of FEAT_CSV2_2:

- ID_AA64PFR0_EL1.CSV2.

For more information, see:

- For AArch64, ‘Restrictions on the effects of speculation’.

FEAT_CSV2_3, Cache Speculation Variant 2 version 3

FEAT_CSV2_3 introduces a mechanism to identify if hardware cannot disclose information about whether prediction resources based on branch history trained in one hardware described context can control speculative execution in a different hardware described context.

FEAT_CSV2_3 is OPTIONAL from Armv8.0.

If FEAT_CSV2_3 is implemented, then FEAT_CSV2_2 is implemented.

The following field identifies the presence of FEAT_CSV2_3:

- ID_AA64PFR0_EL1.CSV2.

For more information, see:

- For AArch64, ‘Restrictions on the effects of speculation’.

FEAT_Crypto, Cryptographic Extension

The Arm Cryptographic Extension provides instructions for the acceleration of encryption and decryption. The presence of the Cryptographic Extension in an implementation is subject to export license controls.

The Cryptographic Extension is an extension of the SIMD support and operates on the vector register file. It also provides multiply instructions that operate on long polynomials.

If FEAT_Crypto is implemented, then FEAT_AES or FEAT_SHA1 is implemented.

In an Armv8.2 implementation, if FEAT_Crypto is implemented, FEAT_PMULL, FEAT_SHA256, FEAT_SM3, or FEAT_SM4 is implemented.

For more information, see:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The Cryptographic Extension in AArch64 state.

FEAT_DoubleLock, Double Lock

FEAT_DoubleLock is the mnemonic used for the OS Double Lock.

FEAT_DoubleLock is OPTIONAL from Armv8.0.

The following field identifies the presence of FEAT_DoubleLock:

- ID_AA64DFR0_EL1.DoubleLock.

FEAT_EL0, Support for execution at EL0

This feature is mandatory.

FEAT_EL1, Support for execution at EL1

This feature is mandatory.

FEAT_EL2, Support for execution at EL2

This feature is mandatory.

FEAT_ETMv4, Embedded Trace Macrocell version 4

FEAT_ETMv4 indicates support for the Embedded Trace Macrocell architecture ETMv4.

FEAT_ETMv4 is OPTIONAL from Armv8.0.

If FEAT_ETMv4 is implemented, then FEAT_TRC_SR or FEAT_TRC_EXT is implemented.

For more information, see the Arm® Embedded Trace Macrocell Architecture Specification, ETMv4(ARM IHI 0064).

FEAT_FP, Floating Point extensions

FEAT_FP includes support for single-precision and double-precision floating-point types. All Armv8-A systems that support standard operating systems with rich application environments also provide hardware support for floating-point instructions.

FEAT_FP is OPTIONAL from Armv8.0.

If FEAT_FP is implemented, then FEAT_AdvSIMD is implemented.

The following fields identify the presence of FEAT_FP:

- ID_AA64PFR0_EL1.FP.
- EDPFR.FP.

For more information, see:

- Supported data types.
- Floating-point support.
- A64 Advanced SIMD and Floating-point Instruction Descriptions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_IVIPT, The IVIPT Extension

FEAT_IVIPT describes any permitted instruction cache implementation. This includes Virtual Index, Physical tag (VIPT) cache policy and Physical Index, Physical Tag (PIPT) cache policy.

FEAT_IVIPT is mandatory from Armv8.0.

.

For more information, see:

- Instruction caches.
- ‘Instruction caches’.

FEAT_LittleEnd, Support for little-endian at EL1 and above

FEAT_LittleEnd is OPTIONAL from Armv8.0.

If FEAT_LittleEnd is implemented, then FEAT_LittleEndEL0 is implemented.

If FEAT_MixedEnd is implemented, then FEAT_LittleEnd is implemented.

FEAT_LittleEndEL0, Support for little-endian at EL0

FEAT_LittleEndEL0 is OPTIONAL from Armv8.0.

If FEAT_MixedEndEL0 is implemented, then FEAT_LittleEndEL0 is implemented.

FEAT_MixedEnd, Mixed-endian support

FEAT_MixedEnd provides support for mixed-endian configuration.

FEAT_MixedEnd is OPTIONAL from Armv8.0.

If FEAT_MixedEnd is implemented, then FEAT_MixedEndEL0 is implemented.

The following field identifies the presence of FEAT_MixedEnd:

- ID_AA64MMFR0_EL1.BigEnd.

For more information see Mixed-endian support in AArch64.

FEAT_MixedEndEL0, Mixed-endian support at EL0

FEAT_MixedEndEL0 provides support for mixed-endian at EL0.

FEAT_MixedEndEL0 is OPTIONAL from Armv8.0.

The following fields identify the presence of FEAT_MixedEndEL0:

- ID_AA64MMFR0_EL1.BigEndEL0.
- ID_AA64MMFR0_EL1.BigEnd.

For more information see Mixed-endian support in AArch64.

FEAT_PMSAv64, Protected Memory System Architecture v8-64

For more information, see The AArch64 Protected Memory System Architecture

FEAT_PMULL, Advanced SIMD PMULL instructions

FEAT_PMULL provides the AES* instructions that support multiplication of 64-bit polynomials, PMULL, PMULL2.

FEAT_PMULL is OPTIONAL from Armv8.0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If FEAT_PMULL is implemented, then FEAT_AES is implemented.

The following field identifies the presence of FEAT_PMULL:

- ID_AA64ISAR0_EL1.AES.

FEAT_PMUv3, PMU extension version 3

The Performance Monitors Extension, FEAT_PMUv3, is an optional non-invasive debug component.

FEAT_PMUv3 is OPTIONAL from Armv8.0.

The following fields identify the presence of FEAT_PMUv3:

- ID_AA64DFR0_EL1.PMUVer.
- EDDFR.PMUVer.

For more information, see The Performance Monitors Extension.

FEAT_PMUv3_EXT, External interface to the Performance Monitors

FEAT_PMUv3_EXT indicates support for external access to the FEAT_PMUv3 and/or FEAT_PCSRv8p2 registers.

FEAT_PMUv3_EXT is OPTIONAL from Armv8.0.

If FEAT_PMUv3_EXT is implemented, then FEAT_PMUv3 or FEAT_PCSRv8p2 is implemented.

If FEAT_PMUv3_EXT is implemented, then FEAT_PMUv3_EXT32 is implemented.

FEAT_PMUv3_EXT32, 32-bit external interface to the Performance Monitors

FEAT_PMUv3_EXT32 indicates the external Performance Monitors and CoreSight registers are implemented as mostly 32-bit registers.

FEAT_PMUv3_EXT32 is OPTIONAL from Armv8.0.

If FEAT_PMUv3_EXT32 is implemented, then FEAT_PMUv3_EXT is implemented.

The following field identifies the presence of FEAT_PMUv3_EXT32:

- PMDEVARCH.ARCHPART.

For more information, see Recommended External Interface to the Performance Monitors.

FEAT_SHA1, Advanced SIMD SHA1 instructions

FEAT_SHA1 implements the SHA1* instructions.

FEAT_SHA1 is OPTIONAL from Armv8.0.

If FEAT_SHA1 is implemented, then FEAT_Crypto is implemented.

The following field identifies the presence of FEAT_SHA1:

- ID_AA64ISAR0_EL1.SHA1.

FEAT_SHA256, Advanced SIMD SHA256 instructions

FEAT_SHA256 implements the SHA256* instructions.

FEAT_SHA256 is OPTIONAL from Armv8.0.

If FEAT_SHA256 is implemented, then FEAT_SHA1 is implemented.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The following field identifies the presence of FEAT_SHA256:

- ID_AA64ISAR0_EL1.SHA2.

FEAT_Secure, Support for Secure state

FEAT_Secure adds support for Secure state.

FEAT_Secure is OPTIONAL from Armv8.0.

FEAT_SpecSEI, SError interrupt exceptions from speculative reads of memory

Describes whether the PE can generate SError interrupt exceptions from speculative reads of memory, including speculative instruction fetches.

FEAT_SpecSEI is OPTIONAL from Armv8.0.

The following field identifies the presence of FEAT_SpecSEI:

- ID_AA64MMFR1_EL1.SpecSEI.

FEAT_TGran16K, Support for 16KB memory translation granule size at stage 1

FEAT_TGran16K is OPTIONAL from Armv8.0.

The following field identifies the presence of FEAT_TGran16K:

- ID_AA64MMFR0_EL1.TGran16.

FEAT_TGran4K, Support for 4KB memory translation granule size at stage 1

FEAT_TGran4K is OPTIONAL from Armv8.0.

The following field identifies the presence of FEAT_TGran4K:

- ID_AA64MMFR0_EL1.TGran4.

FEAT_TGran64K, Support for 64KB memory translation granule size at stage 1

FEAT_TGran64K is OPTIONAL from Armv8.0.

The following field identifies the presence of FEAT_TGran64K:

- ID_AA64MMFR0_EL1.TGran64.

FEAT_TRC_EXT, Trace external registers

FEAT_TRC_EXT indicates support for external access to the ETMv4 or ETE registers.

FEAT_TRC_EXT is OPTIONAL from Armv8.0.

If FEAT_TRC_EXT is implemented, then FEAT_ETMv4 is implemented.

FEAT_TRC_SR, Trace System registers

FEAT_TRC_SR indicates support for System registers for ETMv4 or ETE.

FEAT_TRC_SR is OPTIONAL from Armv8.0.

If FEAT_TRC_SR is implemented, then FEAT_ETMv4 is implemented.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The following fields identify the presence of FEAT_TRC_SR:

- ID_AA64DFR0_EL1.TraceVer.
- EDDFR.TraceVer.

FEAT_nTLBPA, Intermediate caching of translation table walks

FEAT_nTLBPA introduces a mechanism to identify if the intermediate caching of translation table walks does not include non-coherent caches of previous valid translation table entries since the last completed TLBI applicable to the PE.

FEAT_nTLBPA is OPTIONAL from Armv8.0.

The following field identifies the presence of FEAT_nTLBPA:

- ID_AA64MMFR1_EL1.nTLBPA.

For more information, see:

- TLB maintenance.

A2.2.1.1 Features added to the Armv8.0 extension in later releases

- FEAT_CSV2.
- FEAT_CSV3.
- FEAT_DGH.
- FEAT_RAS.
- FEAT_SB.
- FEAT_SPECRES.
- FEAT_SSBS2.
- FEAT_SSBS.

A2.2.2 The Armv8.1 architecture extension

The Armv8.1 architecture extension is an extension to Armv8.0. It adds mandatory and optional architectural features. Some features must be implemented together. An implementation is Armv8.1 compliant if all of the following apply:

- It is Armv8.0 compliant.
- It includes all of the Armv8.1 architectural features that are mandatory.

An Armv8.1 compliant implementation can additionally include:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Armv8.1 features that are optional.
- Any arbitrary subset of the architectural features of Armv8.2, subject only to those constraints that require that certain features be implemented together.

FEAT_Debugv8p1, Debug v8.1

FEAT_Debugv8p1 is OPTIONAL from Armv8.0.

The following fields identify the presence of FEAT_Debugv8p1:

- ID_AA64DFR0_EL1.DebugVer.
- EDDEVARCH.ARCHVER.

FEAT_HAFDBS, Hardware management of the Access flag and dirty state

In Armv8.0, all updates to the translation tables are performed by software. From Armv8.1, for the VMSAv8-64 translation regimes only, hardware can perform updates to the translation tables in two contexts:

- Hardware management of the Access flag.
- Hardware management of dirty state, with updates to a dirty state in the translation tables.

The dirty state is introduced in Armv8.1.

Hardware management of dirty state can be enabled only when hardware management of the Access flag is also enabled.

FEAT_HAFDBS is OPTIONAL from Armv8.0.

The following field identifies the presence of FEAT_HAFDBS:

- ID_AA64MMFR1_EL1.HAFDBS.

For more information, see:

- Hardware management of the Access flag.
- Hardware management of the dirty state.

FEAT_HPDS, Hierarchical permission disables in translations tables

FEAT_HPDS introduces the facility to disable the hierarchical permissions, APTable, PXNTable, and UXNTable, in the translation tables. This disable has no effect on the NSTable bit.

FEAT_HPDS is OPTIONAL from Armv8.0.

FEAT_HPDS is mandatory from Armv8.1.

The following field identifies the presence of FEAT_HPDS:

- ID_AA64MMFR1_EL1.HPDS.

This feature is added only to the VMSAv8-64 translation regimes. Armv8.2 extends this to the AArch32 translation regimes, see FEAT_AA32HPD.

FEAT_LSE, Large System Extensions

FEAT_LSE introduces a set of atomic instructions:

- Compare and Swap instructions, `CAS` and `CASP`.
- Atomic memory operation instructions, LD<OP> and ST<OP>, where <OP> is one of ADD, CLR, EOR, SET, SMAX, SMIN, UMAX, and UMIN.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Swap instruction, SWP.

FEAT_LSE is OPTIONAL from Armv8.0.

FEAT_LSE is mandatory from Armv8.1.

The following field identifies the presence of FEAT_LSE:

- ID_AA64ISAR0_EL1.Atomic.

For more information, see:

- 'Atomic integer memory operations'.
- 'Swap operations'.
- 'Compare and Swap operations'.

FEAT_PAN, Privileged access never

FEAT_PAN introduces a bit to PSTATE. When the value of this PAN state bit is 1, any privileged data access from EL1, or EL2 when the Effective value of HCR_EL2.E2H is 1, to a virtual memory address that is accessible to data accesses at EL0, generates a Permission fault.

FEAT_PAN is OPTIONAL from Armv8.0.

FEAT_PAN is mandatory from Armv8.1.

The following field identifies the presence of FEAT_PAN:

- ID_AA64MMFR1_EL1.PAN.

For more information, see:

- PSTATE.PAN.
- 'About the PAN bit'.

FEAT_PMUv3p1, Armv8.1 PMU extensions

FEAT_PMUv3p1 introduces the following:

- The event number space is extended to 16 bits.
- The HPMD bit is added to MDCR_EL2. This bit disables event counting at EL2.
- The STALL_FRONTEND and STALL_BACKEND events are required to be implemented.

FEAT_PMUv3p1 is OPTIONAL from Armv8.0.

In an Armv8.1 implementation, if FEAT_PMUv3 is implemented, FEAT_PMUv3p1 is implemented.

If FEAT_PMUv3p1 is implemented, then FEAT_PMUv3 is implemented.

The following fields identify the presence of FEAT_PMUv3p1:

- ID_AA64DFR0_EL1.PMUVer.
- EDDFR.PMUVer.

For more information, see 'Required events'.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_RDM, Advanced SIMD rounding double multiply-accumulate instructions

FEAT_RDM introduces Rounding Double Multiply Add/Subtract Advanced SIMD instructions. For more information, see:

For the A64 instruction set

- SQRDMLAH (by element).
- SQRDMLAH (vectors).
- SQRDMLSH (by element).
- SQRDMLSH (vector).

FEAT_RDM is OPTIONAL from Armv8.0.

In an Armv8.1 implementation, if FEAT_AdvSIMD is implemented, FEAT_RDM is implemented.

The following field identifies the presence of FEAT_RDM:

- ID_AA64ISAR0_EL1.RDM.

FEAT_VMID16, 16-bit VMID

In an Armv8.1 implementation, when EL2 is using AArch64, the virtual machine identifier (VMID) size is an IMPLEMENTATION DEFINED choice of 8 bits or 16 bits.

FEAT_VMID16 is OPTIONAL from Armv8.0.

The following field identifies the presence of FEAT_VMID16:

- ID_AA64MMFR1_EL1.VMIDBits.

For more information, see ‘VMID size’.

A2.2.2.1 Features added to the Armv8.1 extension in later releases

- FEAT_DPB2.
- FEAT_DotProd.
- FEAT_FHM.
- FEAT_FlagM.

A2.2.3 The Armv8.2 architecture extension

The Armv8.2 architecture extension is an extension to Armv8.1. It adds mandatory and optional architectural features. Some features must be implemented together. An implementation is Armv8.2 compliant if all of the following apply:

- It is Armv8.1 compliant.
- It includes all of the Armv8.2 architectural features that are mandatory.

An Armv8.2 compliant implementation can additionally include:

- Armv8.2 features that are optional.
- Any arbitrary subset of the architectural features of Armv8.3, subject only to those constraints that require that certain features be implemented together.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_ASMv8p2, Armv8.2 changes to the A64 ISA

FEAT_ASMv8p2 introduces the BFC instruction to the A64 instruction set as an alias of BFM. It also requires that the BFC instruction and the A64 pseudo-instruction REV64 are implemented by assemblers.

In Armv8.0 and Armv8.1, the A64 pseudo-instruction REV64 is OPTIONAL.

FEAT_ASMv8p2 is OPTIONAL from Armv8.1.

FEAT_ASMv8p2 is mandatory from Armv8.2.

For more information, see:

- BFC.
- REV64.

FEAT_DPB, DC CVAP instruction

FEAT_DPB introduces a mechanism to identify and manage persistent memory locations in a shared memory hierarchy, including adding the DC CVAP instruction.

FEAT_DPB is OPTIONAL from Armv8.1.

FEAT_DPB is mandatory from Armv8.2.

The following field identifies the presence of FEAT_DPB:

- ID_AA64ISAR1_EL1.DPB.

For more information, see Memory hierarchy.

FEAT_Debugv8p2, Debug v8.2

FEAT_Debugv8p2 provides the following changes:

- If the Core power domain is powered up and DoubleLockStatus() == TRUE, EDPRSR.{DLK,SPD,PU} is only permitted to read {UNKNOWN, 0, 0}.
- The definition of Exception Catch debug events is extended to include reset entry.
- All CONSTRAINED UNPREDICTABLE cases that generate Exception Catch debug events are removed.
- Controls are added to EDECCR to control Exception Catch debug event generation on exception return.
- All IMPLEMENTATION DEFINED control of external debug accesses to OSLAR_EL1 is removed.
- ExternalSecureNoninvasiveDebugEnabled() cannot override software controls of counting attributable events in Secure state.

FEAT_Debugv8p2 is OPTIONAL from Armv8.1.

FEAT_Debugv8p2 is mandatory from Armv8.2.

If FEAT_Debugv8p2 is implemented, then FEAT_Debugv8p1 is implemented.

The following fields identify the presence of FEAT_Debugv8p2:

- ID_AA64DFR0_EL1.DebugVer.
- EDDEVARCH.ARCHVER.

For more information, see:

- Exception Catch debug event.
- EDPRSR.{DLK, SPD, PU} and the Core power domain.
- ‘Interaction with EL3’.
- External access disabled.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_FP16, Half-precision floating-point data processing

FEAT_FP16 supports:

- Half-precision data-processing instructions for Advanced SIMD and floating-point.

FPCR.FZ16 enables flushing of denormalized numbers to zero for half-precision data-processing instructions.

FEAT_FP16 is OPTIONAL from Armv8.2.

In an Armv8.4 implementation, if FEAT_FHM is not implemented, then FEAT_FP16 is not implemented.

In an Armv8.2 implementation, if FEAT_FHM is implemented, FEAT_FP16 is implemented.

The following fields identify the presence of FEAT_FP16:

- ID_AA64PFR0_EL1.FP.
- ID_AA64PFR0_EL1.AdvSIMD.

For more information, see:

- Half-precision floating-point formats.
- Flushing denormalized numbers to zero.
- Modified immediate constants in A64 floating-point instructions.

FEAT_HPDS2, Hierarchical permission disables

FEAT_HPDS2 provides a mechanism to allow operating systems or hypervisors to make up to four bits of Translation Table final-level descriptors available for IMPLEMENTATION DEFINED hardware use.

FEAT_HPDS2 is OPTIONAL from Armv8.1.

If FEAT_HPDS2 is implemented, then FEAT_HPDS is implemented.

The following field identifies the presence of FEAT_HPDS2:

- ID_AA64MMFR1_EL1.HPDS.

For more information, see:

- ‘Page Based Hardware attributes’.
- ‘Attribute fields in VMSAv8-32 Long-descriptor translation table format descriptors’.

FEAT_IESB, Implicit Error Synchronization event

FEAT_IESB introduces control for an implicit error synchronization event at exception entry and return.

The implicit error synchronization events affect the same synchronizable asynchronous events that are synchronized by the ESB instruction. This means that if an implementation has either no sources of SError exceptions, or all SError exceptions are not synchronizable errors, then the controls added by this feature have no effect.

FEAT_IESB is OPTIONAL from Armv8.1.

If FEAT_IESB is implemented, then FEAT_RAS is implemented.

The following field identifies the presence of FEAT_IESB:

- ID_AA64MMFR2_EL1.IESB.

For more information, see Error synchronization event.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_LPA, Large PA and IPA support

FEAT_LPA:

- Allows a larger physical address (PA) and intermediate physical address (IPA) space of up to 52 bits when using the 64KB translation granule.
- Allows a level 1 block size where the block covers a 4TB address range for the 64KB translation granule if the implementation support 52 bits of PA.

FEAT_LPA is OPTIONAL from Armv8.1.

The following field identifies the presence of FEAT_LPA:

- ID_AA64MMFR0_EL1.PARange.

For more information about FEAT_LPA, see:

- Implemented physical address size.
- Output address size configuration.
- Intermediate physical address size configuration
- Translation using the 64KB granule
- 'Table descriptor format'
- Block descriptor and Page descriptor formats

FEAT_LVA, Large VA support

FEAT_LVA supports a *virtual address* (VA) space for each translation table base register of up to 52 bits when using VMSAv8-64 translation format with the 64KB translation granule.

FEAT_LVA is OPTIONAL from Armv8.1.

The following field identifies the presence of FEAT_LVA:

- ID_AA64MMFR2_EL1.VARange.

For more information about FEAT_LVA, see:

- Supported virtual address ranges.
- Input address size configuration.
- Translation using the 64KB granule.

FEAT_PAN2, AT S1E1R and AT S1E1W instruction variants affected by PSTATE.PAN

FEAT_PAN2 introduces variants of the AT S1E1R and AT S1E1W instructions. These instructions factor in the PSTATE.PAN bit when determining whether or not the location will generate a Permission fault for a privileged access, as is reported in the PAR. For more information, see:

For the AArch64 System instructions:

- AT S1E1RP, Address Translate Stage 1 EL1 Read PAN.
- AT S1E1RP, Address Translate Stage 1 EL1 Write PA.

FEAT_PAN2 is OPTIONAL from Armv8.1.

FEAT_PAN2 is mandatory from Armv8.2.

If FEAT_PAN2 is implemented, then FEAT_PAN is implemented.

The following field identifies the presence of FEAT_PAN2:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- ID_AA64MMFR1_EL1.PAN.

For more information, see:

- Address translation instructions.
- Address translation instructions.
- ATS1C**, Address translation stage 1, current security state.
- Encoding and availability of the address translation instructions.

FEAT_PCSRv8p2, PC Sample-based Profiling Extension

FEAT_PCSRv8p2 moves the control and implementation of the OPTIONAL PC Sample-based Profiling Extension from the ED*SR Debug registers to PM*SR registers in the Performance Monitors address space.

FEAT_PCSRv8p2 is OPTIONAL from Armv8.1.

The following field identifies the presence of FEAT_PCSRv8p2:

- PMDEVID.PCSample.

For more information, see The PC Sample-based Profiling Extension.

FEAT_RAS, Reliability, Availability and Serviceability (RAS) Extension

FEAT_RAS introduces the following:

- The Error Synchronization Barrier (ESB) instruction for the A32, T32, and A64 instruction sets.
- Error records.
- The Error synchronization event.
- Additional syndrome information on External Aborts.

FEAT_RAS is OPTIONAL from Armv8.0.

FEAT_RAS is mandatory from Armv8.2.

The following field identifies the presence of FEAT_RAS:

- ID_AA64PFR0_EL1.RAS.

For more information, see:

- Reliability, Availability, and Serviceability.
- RAS PE Architecture.
- *Arm® Reliability Availability and Serviceability (RAS) System Architecture, for A-profile architecture* (ARM IHI 0100).

FEAT_SHA3, Advanced SIMD SHA3 instructions

FEAT_SHA3 introduces Advanced SIMD instructions that support SHA3 functionality.

FEAT_SHA3 is OPTIONAL from Armv8.1.

If FEAT_SHA3 is implemented, then FEAT_Crypto is implemented.

If FEAT_SHA3 is implemented, then FEAT_SHA256 and FEAT_SHA1 are implemented.

The following field identifies the presence of FEAT_SHA3:

- ID_AA64ISAR0_EL1.SHA3.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For more information, see FEAT_SHA3, SHA3 functionality.

FEAT_SHA512, Advanced SIMD SHA512 instructions

FEAT_SHA512 introduces Advanced SIMD instructions that support SHA2-512 functionality.

FEAT_SHA512 is OPTIONAL from Armv8.1.

If FEAT_SHA512 is implemented, then FEAT_Crypto is implemented.

If FEAT_SHA512 is implemented, then FEAT_SHA256 and FEAT_SHA1 are implemented.

The following field identifies the presence of FEAT_SHA512:

- ID_AA64ISAR0_EL1.SHA2.

For more information, see FEAT_SHA512, SHA2-512 functionality.

FEAT_SM3, Advanced SIMD SM3 instructions

FEAT_SM3 introduces Advanced SIMD instructions that support the Chinese cryptography algorithm SM3.

FEAT_SM3 is OPTIONAL from Armv8.1.

If FEAT_SM3 is implemented, then FEAT_Crypto is implemented.

The following field identifies the presence of FEAT_SM3:

- ID_AA64ISAR0_EL1.SM3.

For more information, see FEAT_SM3, SM3 functionality.

FEAT_SM4, Advanced SIMD SM4 instructions

FEAT_SM4 introduces Advanced SIMD instructions that support the Chinese cryptography algorithm SM4.

FEAT_SM4 is OPTIONAL from Armv8.1.

If FEAT_SM4 is implemented, then FEAT_Crypto is implemented.

The following field identifies the presence of FEAT_SM4:

- ID_AA64ISAR0_EL1.SM4.

For more information, see FEAT_SM4, SM4 functionality.

FEAT_TTCNP, Translation table Common not private translations

FEAT_TTCNP permits multiple PEs in the same Inner Shareable domain to use the same translation tables for a given stage of address translation.

This facility is available for all VMSAv8-64 translation regimes and for VMSAv8-32 translation stages that use the Long descriptor Translation Table format.

FEAT_TTCNP is OPTIONAL from Armv8.1.

FEAT_TTCNP is mandatory from Armv8.2.

The following field identifies the presence of FEAT_TTCNP:

- ID_AA64MMFR2_EL1.CnP.

For more information, see:

- Common not private translations.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- ‘Common not private translations in VMSAv8-32’.

FEAT_UAO, Unprivileged Access Override control

FEAT_UAO introduces the UAO bit to PSTATE. When the Effective value of UAO is 1, if memory accesses that are described as unprivileged are made by the unprivileged memory access instructions, they are privileged.

FEAT_UAO is OPTIONAL from Armv8.1.

FEAT_UAO is mandatory from Armv8.2.

The following field identifies the presence of FEAT_UAO:

- ID_AA64MMFR2_EL1.UAO.

For more information, see:

- PSTATE.UAO.
- Load/store unprivileged.
- UAO.

FEAT_XNX, Translation table stage 2 Unprivileged Execute-never

FEAT_XNX extends the stage 2 translation table access permissions to provide control of whether memory is executable at EL0 independent of whether it is executable at EL1.

This facility is available for stage 2 translation stages in VMSAv8-64 and VMSAv8-32.

FEAT_XNX is OPTIONAL from Armv8.1.

FEAT_XNX is mandatory from Armv8.2.

The following field identifies the presence of FEAT_XNX:

- ID_AA64MMFR1_EL1.XNX.

For more information, see:

- ‘Stage 2 instruction execution using Direct permissions’.

A2.2.3.1 Features added to the Armv8.2 extension in later releases

- FEAT_LRCPC2.
- FEAT_LSE2.
- FEAT_PAuth2.
- FEAT_RASSAv1p1.
- FEAT_RASv1p1.

A2.2.4 The Armv8.3 architecture extension

The Armv8.3 architecture extension is an extension to Armv8.2. It adds mandatory and optional architectural features. Some features must be implemented together. An implementation is Armv8.3 compliant if all of the following apply:

- It is Armv8.2 compliant.
- It includes all of the Armv8.3 architectural features that are mandatory.

An Armv8.3 compliant implementation can additionally include:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Armv8.3 features that are optional.
- Any arbitrary subset of the architectural features of Armv8.4, subject only to those constraints that require that certain features be implemented together.

FEAT_CCIDX, Extended cache index

FEAT_CCIDX introduces the following registers to allow caches to be described with greater numbers of sets and greater associativity:

- A 64-bit format of CCSIDR_EL1.
- CCSIDR2_EL1.

FEAT_CCIDX is OPTIONAL from Armv8.2.

The following field identifies the presence of FEAT_CCIDX:

- ID_AA64MMFR2_EL1.CCIDX.

For more information, see:

- Possible formats of the Cache Size Identification Register, CCSIDR_EL1.

FEAT_CONSTPACFIELD, PAC algorithm enhancement

FEAT_CONSTPACFIELD introduces functionality that permits an implementation with pointer authentication to use the value of bit[55] in the virtual address to determine the size of the PAC field when adding a PAC to the virtual address, even when the top byte is not being ignored.

FEAT_CONSTPACFIELD is OPTIONAL from Armv8.2.

If FEAT_CONSTPACFIELD is implemented, then FEAT_PAuth2 is implemented.

The following field identifies the presence of FEAT_CONSTPACFIELD:

- ID_AA64ISAR2_EL1.PAC_frac.

For more information, see PAC field.

FEAT_DoPD, Debug over Powerdown

FEAT_DoPD provides a debug programmers' model where all debug and PMU registers are in the Core power domain and all CTI registers are in the Debug power domain. Power control is provided by an OPTIONAL external powerup request mechanism.

When FEAT_DoPD is implemented:

- The optional Software Lock is not implemented by the architecturally defined debug components in the PE Core power domain.
- If an ETMv4 trace unit is implemented, the ETM must implement:
  - ETMv4.2 or later.
  - The Unified Power Domain Model.
- If FEAT_ETE is implemented, the trace unit always implements a single power domain.

FEAT_DoPD is OPTIONAL from Armv8.2.

If FEAT_DoPD is implemented, then FEAT_DoubleLock is not implemented.

If FEAT_DoPD is implemented, then FEAT_Debugv8p2 is implemented.

The following field identifies the presence of FEAT_DoPD:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- EDDEVID.DebugPower.

For more information, see Debug Reset and Powerdown Support.

FEAT_EPAC, Enhanced pointer authentication

FEAT_EPAC introduces functionality that permits setting the Pointer Authentication Code (PAC) field to 0 on performing a PAC operation on a non-canonical address.

FEAT_EPAC is OPTIONAL from Armv8.2.

If FEAT_EPAC is implemented, then FEAT_PAuth is implemented.

If FEAT_EPAC is implemented, then FEAT_PAuth2 is not implemented.

The following fields identify the presence of FEAT_EPAC:

- ID_AA64ISAR1_EL1.APA.
- ID_AA64ISAR1_EL1.API.
- ID_AA64ISAR2_EL1.APA3.

For more information, see Pointer authentication.

FEAT_FCMA, Floating-point complex number instructions

FEAT_FCMA introduces instructions for floating-point multiplication and addition of complex numbers.

FEAT_FCMA is OPTIONAL from Armv8.2.

In an Armv8.3 implementation, if FEAT_FP is implemented, FEAT_FCMA is implemented.

If FEAT_FCMA is implemented, then FEAT_FP is implemented.

The following field identifies the presence of FEAT_FCMA:

- ID_AA64ISAR1_EL1.FCMA.

For more information, see:

- AArch64 Advanced SIMD complex number arithmetic instructions.

FEAT_FPAC, Faulting on AUT* instructions

FEAT_FPAC introduces faulting on an AUT* instruction.

FEAT_FPAC is added as a further extension to FEAT_PAuth2.

FEAT_FPAC is OPTIONAL from Armv8.2.

If FEAT_FPAC is implemented, then FEAT_PAuth2 is implemented.

The following fields identify the presence of FEAT_FPAC:

- ID_AA64ISAR1_EL1.APA.
- ID_AA64ISAR1_EL1.API.
- ID_AA64ISAR2_EL1.APA3.

For more information, see Faulting on pointer authentication.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_FPACCOMBINE, Faulting on combined pointer authentication instructions

FEAT_FPACCOMBINE introduces faulting on the combined instructions that perform pointer authentication.

FEAT_FPACCOMBINE is added as a further extension to FEAT_FPAC.

FEAT_FPACCOMBINE is OPTIONAL from Armv8.2.

If FEAT_FPACCOMBINE is implemented, then FEAT_FPAC is implemented.

The following fields identify the presence of FEAT_FPACCOMBINE:

- ID_AA64ISAR1_EL1.APA.
- ID_AA64ISAR1_EL1.API.
- ID_AA64ISAR2_EL1.APA3.

For more information, see Faulting on pointer authentication.

FEAT_FPACC_SPEC, Speculative behavior of pointer authentication instructions

FEAT_FPACC_SPEC introduces consistent impact of speculation for instructions that perform authentication.

FEAT_FPACC_SPEC is added as a further extension to FEAT_FPACCOMBINE.

FEAT_FPACC_SPEC is OPTIONAL from Armv8.2.

If FEAT_FPACC_SPEC is implemented, then FEAT_FPACCOMBINE is implemented.

The following field identifies the presence of FEAT_FPACC_SPEC:

- ID_AA64MMFR3_EL1.Spec_FPACC.

For more information, see Faulting on pointer authentication.

FEAT_JSCVT, JavaScript conversion instructions

FEAT_JSCVT introduces JavaScript convert instructions that truncate a double-precision value to a 32-bit signed integer, setting the condition flags to indicate whether the converted value was in range.

FEAT_JSCVT is OPTIONAL from Armv8.2.

In an Armv8.3 implementation, if FEAT_FP is implemented, FEAT_JSCVT is implemented.

If FEAT_JSCVT is implemented, then FEAT_FP is implemented.

The following field identifies the presence of FEAT_JSCVT:

- ID_AA64ISAR1_EL1.JSCVT.

For more information, see:

- Floating-point conversion.
- About the A64 Advanced SIMD and floating-point instructions.
- Advanced SIMD and floating-point instructions.
- Floating-point data-processing instructions.

FEAT_LRCPC, Load-Acquire RCpc instructions

FEAT_LRCPC introduces instructions that support the weaker Release Consistency processor consistent (RCpc) model that enables the reordering of a Store-Release followed by a Load-Acquire to a different address.

FEAT_LRCPC is OPTIONAL from Armv8.2.

FEAT_LRCPC is mandatory from Armv8.3.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The following field identifies the presence of FEAT_LRCPC:

- ID_AA64ISAR1_EL1.LRCPC.

For more information, see:

- Load-Acquire, Load-AcquirePC, and Store-Release.
- Load-Acquire/Store-Release.

FEAT_PACIMP, Pointer authentication - IMPLEMENTATION DEFINED algorithm

FEAT_PACIMP permits an IMPLEMENTATION DEFINED cryptographic algorithm to be used for PAC calculation.

FEAT_PACIMP is OPTIONAL from Armv8.2.

If FEAT_PACIMP is implemented, then FEAT_PAuth is implemented.

If FEAT_PACIMP is implemented, then FEAT_PACQARMA5 is not implemented.

If FEAT_PACIMP is implemented, then FEAT_PACQARMA3 is not implemented.

The following fields identify the presence of FEAT_PACIMP:

- ID_AA64ISAR1_EL1.GPI.
- ID_AA64ISAR1_EL1.API.

For more information, see Pointer authentication.

FEAT_PACQARMA3, Pointer authentication - QARMA3 algorithm

FEAT_PACQARMA3 introduces the QARMA3 cryptographic algorithm for PAC calculation.

FEAT_PACQARMA3 is OPTIONAL from Armv8.2.

If FEAT_PACQARMA3 is implemented, then FEAT_PAuth is implemented.

If FEAT_PACQARMA3 is implemented, then FEAT_PACQARMA5 is not implemented.

If FEAT_PACQARMA3 is implemented, then FEAT_PACIMP is not implemented.

The following fields identify the presence of FEAT_PACQARMA3:

- ID_AA64ISAR2_EL1.GPA3.
- ID_AA64ISAR2_EL1.APA3.

For more information, see Pointer authentication.

FEAT_PACQARMA5, Pointer authentication - QARMA5 algorithm

FEAT_PACQARMA5 introduces the QARMA5 cryptographic algorithm for PAC calculation.

FEAT_PACQARMA5 is OPTIONAL from Armv8.2.

If FEAT_PACQARMA5 is implemented, then FEAT_PAuth is implemented.

If FEAT_PACQARMA5 is implemented, then FEAT_PACQARMA3 is not implemented.

If FEAT_PACQARMA5 is implemented, then FEAT_PACIMP is not implemented.

The following fields identify the presence of FEAT_PACQARMA5:

- ID_AA64ISAR1_EL1.GPA.
- ID_AA64ISAR1_EL1.APA.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For more information, see Pointer authentication.

FEAT_PAuth, Pointer authentication

FEAT_PAuth introduces functionality that supports address authentication of the contents of a register before that register is used as the target of an indirect branch, or as a load.

When FEAT_PAuth is implemented, all of the following must be true:

- Exactly one of the PAC algorithms is implemented.
- The PACGA instruction and other Pointer authentication instructions use the same algorithm.

The PAC algorithm features are:

- FEAT_PACQARMA5.
- FEAT_PACIMP.
- FEAT_PACQARMA3.

FEAT_PAuth is OPTIONAL from Armv8.2.

FEAT_PAuth is mandatory from Armv8.3.

If FEAT_PAuth is implemented, then FEAT_PACQARMA5, FEAT_PACIMP, or FEAT_PACQARMA3 is implemented.

For more information, see Pointer authentication.

FEAT_PAuth2, Enhancements to pointer authentication

FEAT_PAuth2 introduces enhanced pointer authentication functionality that changes the mechanism by which a PAC is added to the pointer.

FEAT_PAuth2 is OPTIONAL from Armv8.2.

If FEAT_PAuth2 is implemented, then FEAT_PAuth is implemented.

If FEAT_PAuth2 is implemented, then FEAT_EPAC is not implemented.

The following fields identify the presence of FEAT_PAuth2:

- ID_AA64ISAR1_EL1.APA.
- ID_AA64ISAR1_EL1.API.
- ID_AA64ISAR2_EL1.APA3.

For more information, see Pointer authentication.

A2.2.5 The Armv8.4 architecture extension

The Armv8.4 architecture extension is an extension to Armv8.3. It adds mandatory and optional architectural features. Some features must be implemented together. An implementation is Armv8.4 compliant if all of the following apply:

- It is Armv8.3 compliant.
- It includes all of the Armv8.4 architectural features that are mandatory.

An Armv8.4 compliant implementation can additionally include:

- Armv8.4 features that are optional.
- Any arbitrary subset of the architectural features of Armv8.5, subject only to those constraints that require that certain features be implemented together.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_BBML1, Translation table break-before-make level 1

FEAT_BBML1 provides support for reducing the requirements for following a break-before-make sequence when changing between table or block sizes for a translation.

FEAT_BBML1 is OPTIONAL from Armv8.3.

The following field identifies the presence of FEAT_BBML1:

- ID_AA64MMFR2_EL1.BBM.

For more information, see:

- Block descriptor and Page descriptor formats.
- 'Block translation entry'.
- Support levels for changing block size.

FEAT_BBML2, Translation table break-before-make level 2

FEAT_BBML2 provides support for further reducing the requirements for following a break-before-make sequences when changing between table or block sizes for a translation.

FEAT_BBML2 is OPTIONAL from Armv8.3.

If FEAT_BBML2 is implemented, then FEAT_BBML1 is implemented.

The following field identifies the presence of FEAT_BBML2:

- ID_AA64MMFR2_EL1.BBM.

For more information, see FEAT_BBML1.

FEAT_CNTSC, Generic Counter Scaling

FEAT_CNTSC introduces a scaling register to the memory-mapped counter module that allows the frequency of the counter that is generated to be scaled from the basic frequency reported in the counter ID mechanisms.

FEAT_CNTSC is OPTIONAL from Armv8.3.

The following field identifies the presence of FEAT_CNTSC:

- CNTID.CNTSC.

For more information, see CNTCR.

FEAT_CSV2, Cache Speculation Variant 2

FEAT_CSV2 introduces a mechanism to identify if hardware cannot disclose information about whether any prediction resource, such as branch target, data-value, or cache prefetch, trained in one hardware described context can control speculative execution in a different hardware described context.

FEAT_CSV2 is OPTIONAL from Armv8.0.

The following field identifies the presence of FEAT_CSV2:

- ID_AA64PFR0_EL1.CSV2.

For more information, see:

- For AArch64, 'Restrictions on the effects of speculation'.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_CSV3, Cache Speculation Variant 3

FEAT_CSV3 introduces a mechanism to identify whether data loaded or read from a register under speculation where the data load or register read would not be permitted architecturally, can be used by instructions newer than the load or register read in a manner that allows the value of the inaccessible data to be recovered by code architecturally executed.

FEAT_CSV3 is OPTIONAL from Armv8.0.

The following field identifies the presence of FEAT_CSV3:

- ID_AA64PFR0_EL1.CSV3.

FEAT_DGH, Data Gathering Hint

FEAT_DGH introduces the Data Gathering Hint instruction to the hint space.

FEAT_DGH is OPTIONAL from Armv8.0.

The following field identifies the presence of FEAT_DGH:

- ID_AA64ISAR1_EL1.DGH.

For more information, see Hint instructions.

FEAT_DIT, Data Independent Timing instructions

FEAT_DIT provides a mechanism for software to indicate that a sequence of code is intended to be independent from data values from certain instructions within the sequence.

FEAT_DIT is OPTIONAL from Armv8.3.

FEAT_DIT is mandatory from Armv8.4.

The following field identifies the presence of FEAT_DIT:

- ID_AA64PFR0_EL1.DIT.

For more information, see:

- About PSTATE.DIT.
- About the DIT bit.

FEAT_DPB2, DC CVADP instruction

FEAT_DPB2 allows two levels of cache clean to the Point of Persistence by:

- Redefining Point of Persistence, which changes the scope of DC CVAP.
- Defining a Point of Deep Persistence.
- Adding the DC CVADP System instruction.

FEAT_DPB2 is OPTIONAL from Armv8.1.

If FEAT_DPB2 is implemented, then FEAT_DPB is implemented.

The following field identifies the presence of FEAT_DPB2:

- ID_AA64ISAR1_EL1.DPB.

For more information, see Terminology for Clean, Invalidate, and Clean and Invalidate instructions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_Debugv8p4, Debug v8.4

FEAT_Debugv8p4 provides the following mandatory changes:

- The fields MDCR_EL3.{EPMAD, EDAD} control Non-secure access to the debug and PMU registers. The bus Requester is responsible for other debug authentication.
- The Software Lock is obsolete.
- Non-invasive Debug controls are relaxed.
- Secure and Non-secure views of the debug registers are enabled.

FEAT_Debugv8p4 is OPTIONAL from Armv8.3.

FEAT_Debugv8p4 is mandatory from Armv8.4.

If FEAT_Debugv8p4 is implemented, then FEAT_Debugv8p2 is implemented.

If FEAT_SEL2 is implemented, then FEAT_Debugv8p4 is implemented.

The following fields identify the presence of FEAT_Debugv8p4:

- ID_AA64DFR0_EL1.DebugVer.
- EDDEVARCH.ARCHVER.

For more information, see:

- Definition and constraints of a debugger in the context of external debug.
- External debug interface register access permissions.

FEAT_DotProd, Advanced SIMD dot product instructions

FEAT_DotProd provides instructions to perform a four-way vector dot product of 8-bit integers, accumulating each sum of four products into a 32-bit integer. Each 8-bit input can be treated as a signed or unsigned value.

FEAT_DotProd is OPTIONAL from Armv8.1.

In an Armv8.4 implementation, if FEAT_AdvSIMD is implemented, FEAT_DotProd is implemented.

The following field identifies the presence of FEAT_DotProd:

- ID_AA64ISAR0_EL1.DP.

For more information, see:

- SIMD integer dot product.

FEAT_E0PD, Preventing EL0 access to halves of address maps

FEAT_E0PD prevents access at EL0 to half of the addresses in the memory map.

FEAT_E0PD is OPTIONAL from Armv8.4.

If FEAT_E0PD is implemented, then FEAT_CSV3 is implemented.

The following field identifies the presence of FEAT_E0PD:

- ID_AA64MMFR2_EL1.E0PD.

For more information, see Preventing EL0 access to halves of the address map.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_FHM, Floating-point half-precision to single-precision multiply-add instructions

FEAT_FHM introduces half-precision to single-precision fused multiply-add instructions.

FEAT_FHM is OPTIONAL from Armv8.1.

In an Armv8.4 implementation, if FEAT_FP16 is implemented, FEAT_FHM is implemented.

In an Armv8.2 implementation, if FEAT_FHM is implemented, FEAT_FP16 is implemented.

The following field identifies the presence of FEAT_FHM:

- ID_AA64ISAR0_EL1.FHM.

For more information, see:

- SIMD arithmetic.
- SIMD by element arithmetic.

FEAT_FlagM, Condition flag manipulation instructions

FEAT_FlagM provides instructions that manipulate the PSTATE.{N,Z,C,V} flags.

FEAT_FlagM is OPTIONAL from Armv8.1.

FEAT_FlagM is mandatory from Armv8.4.

The following field identifies the presence of FEAT_FlagM:

- ID_AA64ISAR0_EL1.TS.

For more information, see Flag manipulation instructions.

FEAT_IDST, ID space trap handling

FEAT_IDST causes all AArch64 read accesses to the feature ID space when exceptions are generated to be reported in ESR_ELx using the EC code `0x18`.

FEAT_IDST is OPTIONAL from Armv8.3.

FEAT_IDST is mandatory from Armv8.4.

The following field identifies the presence of FEAT_IDST:

- ID_AA64MMFR2_EL1.IDS.

FEAT_LRCPC2, Load-Acquire RCpc instructions version 2

FEAT_LRCPC2 provides versions of `LDAPR` and `STLR` with a 9-bit unscaled signed immediate offset.

FEAT_LRCPC2 is OPTIONAL from Armv8.2.

FEAT_LRCPC2 is mandatory from Armv8.4.

If FEAT_LRCPC2 is implemented, then FEAT_LRCPC is implemented.

The following field identifies the presence of FEAT_LRCPC2:

- ID_AA64ISAR1_EL1.LRCPC.

For more information, see:

- ‘Changes to single-copy atomicity in Armv8.4’.
- Load-Acquire/Store-Release.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- A64 instructions that are changed in Debug state.

FEAT_LSE2, Large System Extensions version 2

FEAT_LSE2 introduces changes to single-copy atomicity requirements for loads and stores, and changes to alignment requirements for loads and stores.

FEAT_LSE2 is OPTIONAL from Armv8.2.

FEAT_LSE2 is mandatory from Armv8.4.

The following field identifies the presence of FEAT_LSE2:

- ID_AA64MMFR2_EL1.AT.

For more information, see:

- 'Requirements for single-copy atomicity'.
- Alignment of data accesses.

FEAT_PMUv3p4, Arm8.4 PMU extensions

FEAT_PMUv3p4 introduces:

- PMMIR_EL1

FEAT_PMUv3p4 is OPTIONAL from Armv8.3.

If FEAT_PMUv3p4 is implemented, then FEAT_PMUv3p1 is implemented.

The following fields identify the presence of FEAT_PMUv3p4:

- ID_AA64DFR0_EL1.PMUVer.
- EDDFR.PMUVer.

For more information, see PMU Event Descriptions.

FEAT_RASSAv1p1, RAS version 1.1 System Architecture

FEAT_RASSAv1p1 implements the RAS System Architecture v1.1, including support for the following:

- Additional ERR<n>MISC<m> registers.
- The optional RAS Common Fault Injection Model Extension.

FEAT_RASSAv1p1 is OPTIONAL from Armv8.2.

In an Armv8.4 implementation, if FEAT_RAS is implemented, FEAT_RASSAv1p1 is implemented.

If FEAT_RASv1p1 is implemented, then FEAT_RASSAv1p1 is implemented.

FEAT_RASv1p1, RAS extension v1.1

FEAT_RASv1p1 introduces support for System register access to the following RAS System Architecture v1.1 features:

- Additional ERR<n>MISC<m> registers.
- The optional RAS Common Fault Injection Model Extension.

FEAT_RASv1p1 is OPTIONAL from Armv8.2.

In an Armv8.4 implementation, if FEAT_RAS is implemented, FEAT_RASv1p1 is implemented.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If FEAT_RASv1p1 is implemented, then FEAT_RAS is implemented.

If FEAT_RASv1p1 is implemented, then FEAT_RASSAv1p1 is implemented.

The following fields identify the presence of FEAT_RASv1p1:

- ID_AA64PFR0_EL1.RAS.
- ID_AA64PFR1_EL1.RAS_frac.

For more information, see *Arm® Reliability Availability and Serviceability (RAS) System Architecture, for A-profile architecture* (ARM IHI 0100).

FEAT_S2FWB, Stage 2 forced Write-Back

FEAT_S2FWB reduces the requirement of additional cache maintenance instructions in systems where the data Cacheability attributes used by the Guest operating system are different from those expected by the Hypervisor. If this feature is implemented, there is no meaningful distinction between the Inner and Outer Shareability domains for Normal Cacheable accesses to Conventional memory.

FEAT_S2FWB is OPTIONAL from Armv8.3.

FEAT_S2FWB is mandatory from Armv8.4.

The following field identifies the presence of FEAT_S2FWB:

- ID_AA64MMFR2_EL1.FWB.

For more information, see:

- Block descriptor and Page descriptor formats.
- ‘Stage 2 memory type and Cacheability attributes when FEAT_S2FWB is enabled’.

FEAT_SB, Speculation Barrier

Speculation Barrier FEAT_SB introduces a barrier to control speculation.

FEAT_SB is OPTIONAL from Armv8.0.

The following field identifies the presence of FEAT_SB:

- ID_AA64ISAR1_EL1.SB.

For more information, see:

- Speculation Barrier (SB).
- Barriers and CLREX instructions.

FEAT_SEL2, Secure EL2

FEAT_SEL2 permits EL2 to be implemented in Secure state. When Secure EL2 is enabled, a translation regime is introduced that follows the same format as the other Secure translation regimes.

FEAT_SEL2 is OPTIONAL from Armv8.3.

In an Armv8.4 implementation, if FEAT_Secure is implemented, FEAT_SEL2 is implemented.

If FEAT_SEL2 is implemented, then FEAT_TTST is implemented.

If FEAT_SEL2 is implemented, then FEAT_Secure is implemented.

The following field identifies the presence of FEAT_SEL2:

- ID_AA64PFR0_EL1.SEL2.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For more information, see:

- Security states
- Translation regimes.

FEAT_SPECRES, Speculation restriction instructions

FEAT_SPECRES introduces System instructions that prevent predictions based on information gathered from earlier execution within a particular execution context from affecting the later speculative execution within that context, to the extent that the speculative execution is observable through side channels.

FEAT_SPECRES is OPTIONAL from Armv8.0.

The following field identifies the presence of FEAT_SPECRES:

- ID_AA64ISAR1_EL1.SPECRES.

For more information, see:

- ‘Prediction restriction instructions’.
- ‘Execution, data prediction and prefetching restriction System instructions’.

FEAT_SSBS, Speculative Store Bypass Safe

FEAT_SSBS allows software to indicate whether hardware is permitted to load or store speculatively in a manner that could give rise to a cache timing side channel, which in turn could be used to derive an address from values loaded to a register from memory.

FEAT_SSBS is OPTIONAL from Armv8.0.

The following field identifies the presence of FEAT_SSBS:

- ID_AA64PFR1_EL1.SSBS.

For more information, see:

- AArch64 Speculative Store Bypass Safe.

FEAT_SSBS2, MRS and MSR instructions for SSBS version 2

FEAT_SSBS2 provides controls for the MSR and MRS instructions to read and write the PSTATE.SSBS field.

FEAT_SSBS2 is OPTIONAL from Armv8.0.

If FEAT_SSBS2 is implemented, then FEAT_SSBS is implemented.

The following field identifies the presence of FEAT_SSBS2:

- ID_AA64PFR1_EL1.SSBS.

For more information, see:

- AArch64 Speculative Store Bypass Safe.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_TLBIOS, TLB invalidate instructions in Outer Shareable domain

FEAT_TLBIOS provides TLBI maintenance instructions that extend to the Outer Shareable domain.

FEAT_TLBIOS is OPTIONAL from Armv8.3.

FEAT_TLBIOS is mandatory from Armv8.4.

The following field identifies the presence of FEAT_TLBIOS:

- ID_AA64ISAR0_EL1.TLB.

For more information, see TLB maintenance instructions.

FEAT_TLBIRANGE, TLB invalidate range instructions

FEAT_TLBIRANGE provides TLBI maintenance instructions that apply to a range of input addresses.

FEAT_TLBIRANGE is OPTIONAL from Armv8.3.

FEAT_TLBIRANGE is mandatory from Armv8.4.

If FEAT_TLBIRANGE is implemented, then FEAT_TLBIOS is implemented.

The following field identifies the presence of FEAT_TLBIRANGE:

- ID_AA64ISAR0_EL1.TLB.

For more information, see:

- TLB maintenance instructions.
- TLB maintenance instructions that do not apply to a range of addresses.

FEAT_TRF, Self-hosted Trace extensions

FEAT_TRF introduces control of trace in a self-hosted system through System registers.

The feature provides:

- Control of Exception levels and Security states where trace generation is prohibited.
- Control of whether an offset is used for the timestamp recorded with trace information.
- A context synchronization instruction, TSB CSYNC, that can be used to prevent reordering of trace operation accesses with respect to other accesses of the same System registers.

FEAT_TRF is OPTIONAL from Armv8.3.

In an Armv8.4 implementation, if FEAT_ETMv4 is implemented, FEAT_TRF is implemented.

If FEAT_TRF is implemented, then FEAT_TRC_SR is implemented.

The following fields identify the presence of FEAT_TRF:

- ID_AA64DFR0_EL1.TraceFilt.
- EDDFR.TraceFilt.

For more information on FEAT_TRF, see:

- AArch64 Self-hosted Trace.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_TTL, Translation Table Level

FEAT_TTL provides the TTL field to indicate the level of translation table walk holding the leaf entry for the address that is being invalidated. This field is provided in all TLB maintenance instructions that take a VA or an IPA argument.

FEAT_TTL is OPTIONAL from Armv8.3.

FEAT_TTL is mandatory from Armv8.4.

The following field identifies the presence of FEAT_TTL:

- ID_AA64MMFR2_EL1.TTL.

For more information, see:

- TLB maintenance instructions.
- TLB maintenance instructions that do not apply to a range of addresses.

FEAT_TTST, Small translation tables

FEAT_TTST relaxes the lower limit on the size of translation tables, by increasing the maximum permitted value of the T1SZ and T0SZ fields in TCR_EL1, TCR_EL2, TCR_EL3, VTCR_EL2 and VSTCR_EL2.

FEAT_TTST is OPTIONAL from Armv8.3.

If FEAT_SEL2 is implemented, then FEAT_TTST is implemented.

The following field identifies the presence of FEAT_TTST:

- ID_AA64MMFR2_EL1.ST.

For more information, see:

- Input address size configuration.
- Translation using the 4KB granule.
- Translation using the 16KB granule.
- Translation using the 64KB granule.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Part B
The AArch64 Application Level Architecture

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter B1
The AArch64 Application Level Programmers' Model

This chapter describes the AArch64 application level programmers' model. It contains the following sections:

- About the Application level programmers' model.
- Registers in AArch64 Execution state.
- Process state, PSTATE.
- Software control features and EL0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B1.1 About the Application level programmers' model

This chapter contains the programmers' model information required for application development.

The information in this chapter is distinct from the system information required to service and support application execution under an operating system, or higher level of system software. However, some knowledge of the system information is needed to put the Application level programmers' model into context.

Depending on the implementation choices, the architecture supports multiple levels of execution privilege, indicated by different *Exception levels* that number upwards from EL0 to EL2. EL0 corresponds to the lowest privilege level and is often described as unprivileged. The Application level programmers' model is the programmers' model for software executing at EL0. For more information, see Exception levels.

System software determines the Exception level, and therefore the level of privilege, at which software runs. When an operating system supports execution at both EL1 and EL0, an application usually runs unprivileged at EL0. This:

- Permits the operating system to allocate system resources to an application in a unique or shared manner.
- Provides a degree of protection from other processes, and so helps protect the operating system from malfunctioning software.

This chapter indicates where some system level understanding is necessary, and where relevant it gives a reference to the system level description.

Execution at any Exception level above EL0 is often referred to as privileged execution.

For more information on the system level view of the architecture refer to The AArch64 System Level Programmers' Model.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B1.2 Registers in AArch64 Execution state

The following registers are visible at EL0 using AArch64:

R0-R30 31 general-purpose registers, R0 to R30. Each can be accessed as:

- A 64-bit general-purpose register named X0 to X30.
- A 32-bit general-purpose register named W0 to W30.

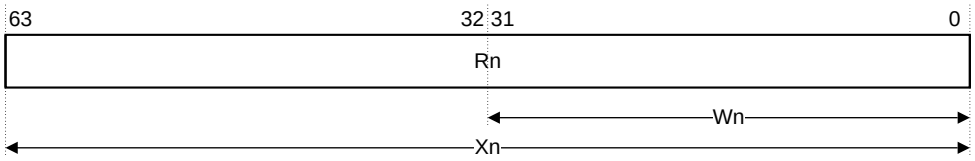


Figure B1-1 General-purpose register naming

The X30 general-purpose register is used as the procedure call link register.

SP A 64-bit dedicated Stack Pointer register. The least significant 32 bits of the stack pointer can be accessed using the register name WSP.

The use of SP as an operand in an instruction, indicates the use of the current stack pointer.

Note

Stack pointer alignment to a 16-byte boundary is configurable at EL1. For more information, see the *Procedure Call Standard for the Arm 64-bit Architecture*.

PC A 64-bit Program Counter holding the address of the current instruction.

Software cannot write directly to the PC. It can be updated only on a branch, exception entry or exception return.

Note

Attempting to execute an A64 instruction that is not word-aligned generates a PC alignment fault, see PC alignment checking.

V0-V31 32 SIMD&FP registers, V0 to V31. Each can be accessed as:

- A 128-bit register named Q0 to Q31.
- A 64-bit register named D0 to D31.
- A 32-bit register named S0 to S31.
- A 16-bit register named H0 to H31.
- An 8-bit register named B0 to B31.

Where the number of bits described by a register name does not occupy an entire SIMD&FP register, it refers to the least significant bits. See Figure B1-2.

For more information about data types and vector formats, see Supported data types.

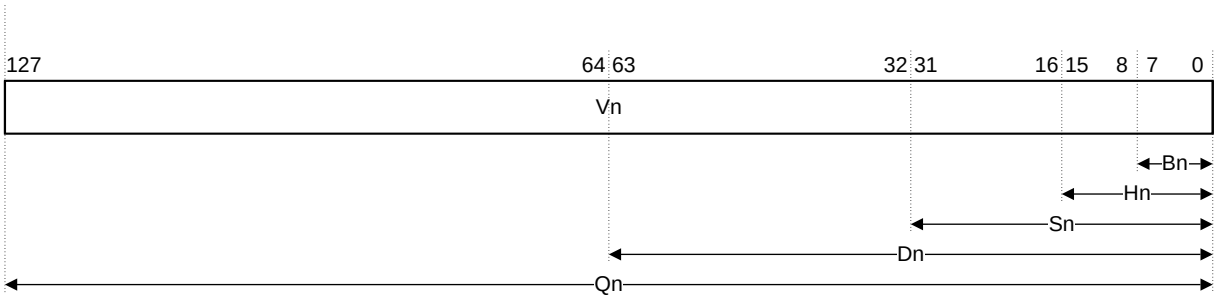


Figure B1-2 SIMD and floating-point register naming

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FPCR, FPSR The FPCR is the floating-point control register. The FPSR is the floating-point status register.

See also:

- System registers.
- Pseudocode description of registers in AArch64 state.
- Registers for instruction processing and exception handling.

B1.2.1 System registers

System registers provide support for execution control, status and general system configuration. The majority of the System registers are not accessible at EL0.

However, some System registers can be configured to allow access from software executing at EL0. Any access from EL0 to a System register without access permission causes the instruction to be trapped. The registers that can be accessed from EL0 are:

Debug registers A Debug Communications Channel is supported by the MDCCSR_EL0, DBGDTR_EL0, DBGDTRRX_EL0, DBGDTRTX_EL0, DLR_EL0 and DSPSR_EL0 registers.

General system control registers

- The CTR_EL0 and DCZID_EL0 registers provide implementation parameters for EL0 cache management support.
- The SCXTNUM_EL0 register provides a number that can be used to separate out different context numbers.
- The POR_EL0 Permission Overlay register determines the unprivileged stage 1 overlay permissions for the EL1&0 and EL2&0 translation regimes.
- The TPIDR_EL0 and TPIDRRO_EL0 registers provide locations where software can store thread identifying information.

Generic timer registers

- Read access to the system counter effective frequency using CNTFRQ_EL0.
- Physical and virtual timer count registers, CNTPCT_EL0, CNTVCT_EL0, CNTPCTSS_EL0 and CNTVCTSS_EL0.
- Physical up-count comparison, down-count value and timer control registers, CNTP_CVAL_EL0, CNTP_TVAL_EL0, and CNTP_CTL_EL0.
- Virtual up-count comparison, down-count value and timer control registers, CNTV_CVAL_EL0, CNTV_TVAL_EL0, and CNTV_CTL_EL0.

Performance Monitors registers

The Performance Monitors Extension provides counters and configuration registers. Software executing at EL1 or a higher Exception level can configure some of these registers to be accessible at EL0.

For more details, see The Performance Monitors Extension.

B1.2.2 Pseudocode description of registers in AArch64 state

The AArch64 pseudocode uses getter and setter accessor functions to read and write register files that are visible at EL0:

- The setter or assignment accessor, for example "`X[num, width] = result`", is used for register writes.
- The getter or non-assignment accessor, "`operand = X[num, width]`", is used for register reads.
- The *num* parameter specifies the register number to be accessed within the named register file, and *width* specifies the bit width of the register access.
- For the general-purpose register `X[]` accessor, register number 31 accesses the zero register, ZR, which reads as zero and ignores writes.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- For other register accessors, such as the SIMD&FP `V[]` accessors, there is no special interpretation of the register number.

Other special AArch64 accessors include:

- The `SP[]` accessors have no parameters and are used to read or write the currently selected 64-bit stack pointer at the current Exception level.
- The `PC64[]` accessor has no parameter list and is used to read the 64-bit Program Counter.
- The `Vpart[]` accessors are used to perform a partial read or write of an Advanced SIMD vector register, using an additional *part* parameter to select which half of the vector register to access.
- The `V[]`, and `Vpart[]` accessors read and write the same underlying vector register file.

Other register accessors are defined as required by other architectural features. See R-profile Architecture Pseudocode, sections aarch64/functions/registers/.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B1.3 Process state, PSTATE

Process state, or PSTATE, is an abstraction of process state information. All of the instruction sets provide instructions that operate on elements of PSTATE.

For the system level view of PSTATE, see Process state, PSTATE in The AArch64 System Level Programmers' Model.

The following PSTATE information is accessible at EL0:

The Condition flags

Flag-setting instructions set these. They are:

N Negative Condition flag. If the result of the instruction is regarded as a two's complement signed integer, the PE sets this to:

- 1 if the result is negative.
- 0 if the result is positive or zero.

Z Zero Condition flag. Set to:

- 1 if the result of the instruction is zero.
- 0 otherwise.

A result of zero often indicates an equal result from a comparison.

C Carry Condition flag. Set to:

- 1 if the instruction results in a carry condition, for example an unsigned overflow that is the result of an addition.
- 0 otherwise.

V Overflow Condition flag. Set to:

- 1 if the instruction results in an overflow condition, for example a signed overflow that is the result of an addition.
- 0 otherwise.

Conditional instructions test the N, Z, C and V Condition flags, combining them with the Condition code for the instruction to determine whether the instruction must be executed. In this way, execution of the instruction is conditional on the result of a previous operation. For more information about conditional execution, see Condition flags and related instructions.

The exception masking bits

D Debug exception mask bit. When EL0 is enabled to modify the mask bits, this bit is visible and can be modified. However, this bit is architecturally ignored at EL0.

A SError mask bit.

I IRQ interrupt mask bit.

F FIQ interrupt mask bit.

For each bit, the values are:

0 Exception not masked.

1 Exception masked.

Access at EL0 using AArch64 state depends on SCTLR_EL1.UMA.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B1.3.1 Accessing PSTATE fields at EL0

At EL0 using AArch64 state, PSTATE fields can be accessed using Special-purpose registers that can be directly read using the MRS instruction and directly written using the MSR (register) instructions. Table B1-1 shows the Special-purpose registers that access the PSTATE fields that hold AArch64 state when the PE is at EL0 using AArch64. All other PSTATE fields do not have direct read and write access at EL0.

Table B1-1 Accessing PSTATE fields at EL0 using MRS and MSR (register)

Special-purpose register	PSTATE fields
NZCV	N, Z, C, V
DAIF	D, A, I, F
SSBS	SSBS
DIT	DIT

Software can also use the MSR (immediate) instruction to directly write to PSTATE.{D, A, I, F, SSBS, DIT}. Table B1-2 shows the MSR (immediate) operands that can directly write to PSTATE.{D, A, I, F, SSBS, DIT} when the PE is at EL0 using AArch64 state.

Table B1-2 Accessing PSTATE.{D, A, I, F, SSBS, DIT} at EL0 using MSR (immediate)

Operand	PSTATE fields	Notes
DAIFSet	D, A, I, F	Directly sets any of the PSTATE.{D,A, I, F} bits to 1
DAIFClr	D, A, I, F	Directly clears any of the PSTATE.{D, A, I, F} bits to 0
SSBS	SSBS	Directly sets the PSTATE.SSBS bit to CRm<0>
DIT	DIT	Directly sets the PSTATE.DIT bit to CRm<0>

Access to the PSTATE.{D, A, I, F} fields at EL0 using AArch64 state depends on SCTLR_EL1.UMA.

Writes to the PSTATE fields have side-effects on various aspects of the PE operation. All of these side-effects, are guaranteed:

- Not to be visible to earlier instructions in the execution stream.
- To be visible to later instructions in the execution stream.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B1.4 Software control features and EL0

The following sections describe the EL0 view of the software control features:

- Exception handling.
- Wait for Interrupt and Wait for Event.
- The YIELD instruction.
- Application level cache management.
- Instructions relating to Debug.
- About PSTATE.DIT.

B1.4.1 Exception handling

In the Arm architecture, an *exception* causes a change of program flow. Execution of an exception handler starts, at an Exception level higher than EL0, from a defined vector that relates to the exception taken.

Exceptions include:

- Interrupts.
- Memory system aborts.
- Exceptions generated by attempting to execute an instruction that is UNDEFINED.
- System calls.
- Secure monitor or Hypervisor traps.
- Debug exceptions.

Most details of exception handling are not visible to application level software, and are described in The AArch64 System Level Programmers' Model.

The `SVC` instruction causes a Supervisor Call exception. This provides a mechanism for unprivileged software to make a system call to an operating system.

The `BRK` instruction generates a Breakpoint Instruction exception. This provides a mechanism for debugging software using debugger executing on the same PE, see Breakpoint Instruction exceptions.

Note

The `BRK` instruction is supported only in the A64 instruction set.

B1.4.2 Wait for Interrupt and Wait for Event

Issuing a `WFI` instruction indicates that no further execution is required until a WFI wakeup event occurs, see Wait for Interrupt mechanism. This permits entry to a low-power state.

Issuing a `WFE` instruction indicates that no further execution is required until a WFE wakeup event occurs, see Wait for Event. This permits entry to a low-power state.

B1.4.3 The YIELD instruction

The `YIELD` instruction provides a hint that the task performed by a thread is of low importance so that it could yield, see YIELD. This mechanism can be used to improve overall performance in a *Symmetric Multithreading* (SMT) or *Symmetric Multiprocessing* (SMP) system.

Examples of when the `YIELD` instruction might be used include a thread that is sitting in a spin-lock, or where the arbitration priority of the snoop bit in an SMP system is modified. The `YIELD` instruction permits binary compatibility between SMT and SMP systems.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The `YIELD` instruction is a `NOP` hint instruction.

The `YIELD` instruction has no effect in a single-threaded system, but developers of such systems can use the instruction to flag its intended use for future migration to a multiprocessor or multithreading system. Operating systems can use `YIELD` in places where a yield hint is wanted, knowing that it will be treated as a NOP if there is no implementation benefit.

B1.4.4 Application level cache management

A small number of cache maintenance instructions can be enabled at EL0 from higher levels of privilege using the SCTLR_EL1 System register. Any access from EL0 to an operation without access permission causes the instruction to be trapped.

About the available operations, see Application level access to functionality related to caches.

B1.4.5 Instructions relating to Debug

Exception handling refers to the `BRK` instruction, which generates a Breakpoint Instruction exception. In addition, in AArch64 state, the `HLT` instruction causes the PE to halt execution and enter Debug state. This provides a mechanism for debugging software using a debugger that is external to the PE, see About External Debug.

B1.4.6 About PSTATE.DIT

A *data-independent-time sequence of code* is the sequence of instructions executed from the first instruction executed after a change of PSTATE.DIT from 0 to 1 through the last instruction executed before a change of PSTATE.DIT from 1 to 0, inclusive. Instructions speculatively executed between the change of PSTATE.DIT from 0 to 1 and the change of PSTATE.DIT from 1 to 0 are considered part of the data-independent-time sequence of code.

A *data-independent-time resource* is any of:

- a general-purpose register other than the zero register (ZR)
- Stack Pointer register
- SIMD&FP register

Note

- The data value prediction applies to all prediction resources that use some form of training to speculate data values as part of an execution.
- The cache allocation applies to all instruction and data caches, and TLB prefetching hardware used by the executing PE that applies to the supplied context.

The context information is passed as a register argument, and is restricted so that:

- Execution of the System instruction at EL0 only applies to the current hardware defined context.
- Execution of the System instruction at EL1only applies to the current VMID and Security state, and does not apply to EL2.
- Execution of the System instruction at EL2 can only apply to the current Security state.

If the System instruction is specified to apply to a combination of Security state and Exception level that is not implemented, or an Exception level which is higher than the Exception level that the System instruction is executed at, then the System instruction is treated as a `NOP`.

When the System instruction is complete and synchronized, no predictions of the restricted type for the affected context are influenced by the execution of the program before the System instruction in a manner that can be observed by the use of any side channels.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

- Prediction restriction System instructions do not require the invalidation of prediction structures so long as the behavior described for completion is met by an implementation.
- Prediction restriction System instructions are permitted to invalidate more prediction information than is defined by the supplied execution context.

These System instructions are guaranteed to be complete following a DSB that covers both read and write behavior on the same PE that executed the original instruction. A subsequent Context synchronization event is required to ensure that the effect of the completion of the instructions is synchronized to the current execution.

In AArch64 state, EL0 access to the System instructions is controlled by SCTLR_EL1.EnRCTX.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B1.5 Memory barriers

Memory barrier is the general term applied to an instruction, or sequence of instructions, that forces synchronization events by a PE with respect to retiring load/store instructions. The memory barriers defined by the Arm architecture provide a range of functionality, including:

- Ordering of load/store instructions.
- Completion of load/store instructions.
- Context synchronization.

The following subsections describe the Arm memory barrier instructions:

- Instruction Synchronization Barrier.
- Data Memory Barrier.
- Speculation Barrier.
- Consumption of Speculative Data Barrier.
- Speculative Store Bypass Barrier.
- Physical Speculative Store Bypass Barrier.
- Data Synchronization Barrier.
- Shareability and access limitations on the data barrier operations.
- Load-Acquire, Load-AcquirePC, and Store-Release.

Note

Depending on the required synchronization, a program might use memory barriers on their own, or it might use them in conjunction with cache maintenance and memory management instructions that in general are available only when software execution is at EL1 or higher.

`DMB` and `DSB` instructions affect reads and writes to the memory system generated by load/store instructions and data or unified cache maintenance instructions being executed by the PE.

B1.5.1 Instruction Synchronization Barrier

An `ISB` instruction ensures that all instructions that come after the `ISB` instruction in program order are fetched from the cache or memory after the `ISB` instruction has completed. Using an `ISB` ensures that the effects of context-changing operations executed before the `ISB` are visible to the instructions fetched after the `ISB` instruction. Examples of context-changing operations that require the insertion of an `ISB` instruction to ensure the effects of the operation are visible to instructions fetched after the `ISB` instruction are:

- Completed cache and TLB maintenance instructions.
- Changes to System registers.

Any context-changing operations appearing in program order after the `ISB` instruction take effect only after the `ISB` has been executed.

The pseudocode function for the operation of an `ISB` is `InstructionSynchronizationBarrier()`.

See also Memory barriers.

B1.5.2 Data Memory Barrier

The `DMB` instruction is a memory barrier instruction that ensures the relative order of memory accesses before the barrier with memory accesses after the barrier. The `DMB` instruction does not ensure the completion of any of the memory accesses for which it ensures relative order.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The full definition of the DMB instruction is covered formally in the Ordering requirements defined by the formal concurrency model and this introduction to the DMB instruction is not intended to contradict that section.

The basic principle of a DMB instruction is to introduce order between memory accesses that are specified to be affected by the DMB options supplied as arguments to the DMB instruction. The DMB instruction ensures that all affected memory accesses by the PE executing the DMB instruction that appear in program order before the DMB instruction and those which originate from a different PE, to the extent required by the DMB options, which have been observed by the PE before the DMB instruction is executed, are observed by each PE, to the extent required by the DMB options, before any affected memory accesses that appear in program order after the DMB instruction are observed by that PE.

The use of a DMB instruction creates order between the Memory effects of instructions as described in the definition of Barrier-ordered-before.

The DMB instruction affects only memory accesses and the operation of data cache and unified cache maintenance instructions, see A64 Cache maintenance instructions. It has no effect on the ordering of any other instructions executing on the PE.

The pseudocode function for the operation of a DMB instruction is `DataMemoryBarrier()`.

B1.5.3 Speculation Barrier

An SB instruction is a memory barrier that prevents speculative execution of instructions until after the barrier has completed when those instructions could be observed through side-channels.

Until the barrier completes, the speculative execution of any instruction appearing later in the program order than the barrier cannot be performed to the extent that such speculation can be observed through side-channels as a result of control flow speculation or data value speculation. An example is speculative allocation into any caching structure where the allocation of that entry could indicate any data value present in memory or in the registers.

- Cannot be performed to the extent that such speculation can be observed through side-channels as a result of control flow speculation or data value speculation.
- Can be performed when predicting that a instruction that could generate an exception does not generate an exception.

The speculative execution of an SB instruction cannot be as a result of any of the following:

- Control flow speculation.
- Data value speculation.
- Can be as a result of predicting that an instruction that could generate an exception does not generate an exception.

An SB instruction can complete when all of the following apply:

- It is known that it is not speculative, or it is speculative only as a result of either:
  - Speculating that an instruction that could generate an exception does not generate an exception.
  - Speculating past the point in the Execution stream where a precise asynchronous exception is taken.
- All the data values generated by instructions appearing in program order before the SB instruction are architecturally resolved, and so are not speculative.

Note

The SB instruction has no effect on the use of prediction resources to predict the instruction stream that is being fetched, so long as the prediction of the instruction stream is not informed by data taken from the register outputs of the speculative execution of instructions appearing in program order after the SB instruction.

B1.5.4 Consumption of Speculative Data Barrier

The CSDB instruction is a memory barrier instruction that controls speculative execution arising from data value prediction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Any instruction, other than a branch instruction, that appears in program order after the CSDB cannot be speculatively executed using the results of any of the following predictions if those predictions come from instructions that appear in program order before the CSDB and have not been architecturally resolved:

- Data value predictions of any instructions.
- `PSTATE.{N,Z,C,V}` predictions of any instructions other than conditional branch instructions appearing in program order before the `CSDB` that have not been architecturally resolved.

Note

For purposes of the definition of CSDB, `PSTATE.{N,Z,C,V}` are not considered data values. This definition permits:

- Control flow speculation before and after the `CSDB` instruction.
- Speculative execution of conditional data processing instructions after the `CSDB` instruction, unless they use the results of data value, or `PSTATE.{N,Z,C,V}` predictions of instructions appearing in program order before the `CSDB` instruction that have not been architecturally resolved.

B1.5.5 Speculative Store Bypass Barrier

The `SSBB` instruction is a memory barrier that prevents speculative loads from bypassing earlier stores to the same virtual address under certain conditions.

The semantics of the Speculative Store Bypass Barrier are:

- When a load to a location appears in program order after the `SSBB` instruction, then the load does not speculatively read an entry earlier in the coherence order for that location than the entry generated by the latest store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store uses the same virtual address as the load.
  - The store appears in program order before the `SSBB` instruction.

B1.5.6 Physical Speculative Store Bypass Barrier

The `PSSBB` instruction is a memory barrier that prevents speculative loads from bypassing earlier stores to the same physical address under certain conditions.

The semantics of the Physical Speculative Store Bypass Barrier are:

- When a load to a location appears in program order after the `PSSBB` instruction, then the load does not speculatively read an entry earlier in the coherence order for that location than the entry generated by the latest store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store appears in program order before the `PSSBB` instruction.

Note

The effect of this barrier applies to accesses to the same location even if they are accessed with different virtual addresses and from different Exception levels.

B1.5.7 Trace Synchronization Barrier

The `TSB` instruction is a barrier instruction that preserves the relative order of accesses to System registers due to trace operations and other accesses to the same registers.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A trace operation is an operation of the trace unit generating trace for an instruction when FEAT_TRF is implemented and enabled.

A `TSB` instruction is not required to execute in program order with respect to other instructions. This includes being reordered with respect to other trace instructions. One or more Context synchronization events are required to ensure that `TSB` instruction is executed in the necessary order.

If trace is generated between a Context synchronization event and a `TSB` operation, these trace operations may be reordered with respect to the `TSB` operation, and therefore may not be synchronized.

The following situations are synchronized using a `TSB` operation:

- A direct write B to a System register is ordered after an indirect read or indirect write of the same register by a trace operation of a traced instruction A, if all of the following are true:
  - A is executed in program order before a Context synchronization event C.
  - C appears in program order before a `TSB` operation T.
  - B is executed in program order after T.
- A direct read B of a System register is ordered after an indirect write to the same register by a trace operation of a traced instruction A if all the following are true:
  - A is executed in program order before a Context synchronization event C1.
  - C1 appears in program order before `TSB` operation T.
  - T is executed in program order before a second Context synchronization event C2.
  - B is executed in program order after C2.

A `TSB` operation is not needed to ensure a direct write B to a System register is ordered before an indirect read or indirect write of the same register by a trace operation of a traced instruction A, if all the following are true:

- A is executed in program order after a Context synchronization event C.
- B is executed in program order before C.

See Synchronization and the Trace Buffer Unit.

The pseudocode function for the operation of a `TSB` instruction is `TraceSynchronizationBarrier()`.

B1.5.8 Data Synchronization Barrier

A `DSB` instruction is a memory barrier that ensures that memory accesses that occur before the `DSB` instruction have completed before the completion of the `DSB` instruction. In doing this, it acts as a stronger barrier than a `DMB` and all ordering that is created by a `DMB` with specific options is also generated by a `DSB` with the same options.

Execution of a `DSB` instruction at EL2 ensures that any memory accesses caused by Speculative translation table walks from the EL1&0 translation regime have been observed.

For more information, see Out-of-context translation regimes.

A `DSB` instruction executed by a PE completes when all of the following are complete for the set of observers in the required shareability domain:

- If the required access types of the `DSB` is reads, all of the following:
  - All Explicit Memory Read Effects generated by instructions in program order before the `DSB`.
  - If FEAT_GCS is implemented, all GCS Memory Read Effects that appear in program order before a GCSB instruction that appears in program order before the `DSB`.
  - If FEAT_MTE2 is implemented, all Implicit Tag Memory Read Effects generated by instructions in program order before the `DSB`.
- If the required access types of the `DSB` is writes, all of the following:
  - All Explicit Memory Write Effects generated by instructions in program order before the `DSB`.
  - If FEAT_GCS is implemented, all GCS Memory Write Effects that appear in program order before a GCSB instruction that appears in program order before the `DSB`.
- If the required access types of the `DSB` is reads and writes, all of the following:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- All Explicit Memory Effects generated by instructions in program order before the DSB.
- All Implicit TTD Memory Effects generated by instructions in program order before the DSB.
- All Implicit Instruction Memory Read Effects generated by instructions in program order before the DSB.
- All cache maintenance instructions in program order before the DSB.
- If FEAT_MTE2 is implemented, all Implicit Tag Memory Read Effects generated by instructions in program order before the DSB.
- If FEAT_SPMU is implemented, all Direct System Register Write Effects to a System PMU register generated by instructions in program order before the DSB.
- If FEAT_SPE is implemented, all PSB CSYNC instructions in program order before the DSB.
- If FEAT_XS is implemented and the DSB has the nXS qualifier, all of the following:
  - All TLBInXS maintenance operations generated by AArch64 TLB maintenance instructions with the nXS qualifier in program order before the DSB.
  - If HCRX_EL2.FnXS is 1, all TLBInXS maintenance operations generated by AArch32 or AArch64 TLB maintenance instructions executed at EL1 in program order before the DSB.
- If FEAT_XS is not implemented or the DSB does not have the nXS qualifier, all of the following:
  - All TLBI maintenance operations generated by TLB maintenance instructions in program order before the DSB.
- If FEAT_HDBSS is implemented and the DSB is executed at EL2 or EL3, all HDBSS Memory Write Effects generated by instructions in program order before the DSB.
- If FEAT_TRBE is implemented, all TSB CSYNC instructions in program order before the DSB.
- All CFP RCTX, COSP RCTX, CPP RCTX, DVP RCTX instructions in program order before the DSB.
- If FEAT_MTE_ASYNC is implemented, all Indirect System Register Write Effects to the Tag Fault Status Register accessible at ELx generated by instructions in program order before the DSB.
- If FEAT_GCS is implemented, all GCS Memory Effects that appear in program order before a GCSB instruction that appears in program order before the DSB.

In addition, no instruction that appears in program order after the DSB instruction can alter any state of the system or perform any part of its functionality until the DSB completes other than:

- Being fetched from memory and decoded, or any change to architectural or microarchitectural state resulting from being fetched or decoded.
- Indirectly or directly reading any of the following, so long as the reading of these resources does not cause a change in architectural state:
  - General-purpose registers
  - SIMD&FP registers
  - The Stack Pointer register
  - The Program Counter
  - Special-purpose registers
  - System registers other than CNTPCTSS_EL0 and CNTVCTSS_EL0

A DSB instruction ordered after a direct write to a System PMU register does not complete until all observers observe the direct write with following ordering requirements:

- EL1 and EL0 memory accesses are ordered only with respect to memory accesses using the same VMID.
- EL2 memory accesses are ordered only with respect to other EL2 memory accesses.

A Context synchronization event is required to create the order between the direct write and the DSB instruction.

If the R-profile architecture is implemented the execution of a DFB at EL2 orders memory accesses irrespective of their Exception level or associated VMID. If executed at EL1 or EL0 the instruction behaves as a DSB SY.

The pseudocode function for the operation of a DSB is DataSynchronizationBarrier().

See also:

- Memory barriers.
- Ordering and completion of TLB maintenance instructions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B1.5.9 Shareability and access limitations on the data barrier operations

The `DMB` and `DSB` instructions take an argument that specifies:

- The shareability domain over which the instruction must operate. This is one of:
  - Full system.
  - Outer Shareable.
  - Inner Shareable.
  - Non-shareable.

  Full system applies to all the observers in the system and, as such, encompasses the Inner and Outer Shareable domains of the processor.

  Note

  The distinction between Full system and Outer Shareable is applicable only for Normal Non-cacheable memory accesses and Device memory accesses.

- The accesses for which the instruction operates. This is one of:
  - Read and write accesses, both before and after the barrier instruction.
  - Write accesses only, before and after the barrier instruction.
  - Read accesses before the barrier instruction, and read and write accesses after the barrier instruction.

    Note

    This form of a `DMB` or `DSB` instruction can be described as a load-load/store barrier.

For more information on whether an access is before or after a barrier instruction, see Data Memory Barrier or Data Synchronization Barrier.

Table B1-3 shows how these options are encoded in the `<option>` field of the instruction:

Table B1-3 Encoding of the `DMB` and `DSB` `<option>` parameter

Accesses		Shareability domain			
Before the barrier	**After the barrier**	**Full system**	**Outer Shareable**	**Inner Shareable**	**Non-shareable**
Reads and writes	Reads and writes	`SY`	`OSH`	`ISH`	`NSH`
Writes	Writes	`ST`	`OSHST`	`ISHST`	`NSHST`
Reads	Reads and writes	`LD`	`OSHLD`	`ISHLD`	`NSHLD`

Note

Table B1-3 specifies the memory ordering requirements of DMB and DSB based on the `<option>` parameter. DSB has additional requirements with respect to execution of instructions appearing in program order after the DSB as described in the section Data Synchronization Barrier. These additional requirements create the effect of additional memory ordering beyond what is specified in the table.

See the instruction descriptions for more information:

- DMB.
- DSB.

Note

`ISB` also supports an optional limitation argument that can contain only one value that corresponds to full system operation, see ISB.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B1.5.10 Load-Acquire, Load-AcquirePC, and Store-Release

Arm provides a set of instructions with Acquire semantics for loads, and Release semantics for stores. These instructions support the *Release Consistency sequentially consistent* (RCsc) model. In addition, FEAT_LRCPC provides Load-AcquirePC instructions. The combination of Load-AcquirePC and Store-Release can be use to support the weaker *Release Consistency processor consistent* (RCpc) model.

The full definitions of the Load-Acquire and Load-AcquirePC instructions are covered formally in the Ordering requirements defined by the formal concurrency model. This introduction to the Load-Acquire and Load-AcquirePC instructions is not intended to contradict that section.

The basic principle of both Load-Acquire and Load-AcquirePC instructions is to introduce order between:

- The memory access generated by the Load-Acquire or Load-AcquirePC instruction.
- The memory accesses appearing in program order after the Load-Acquire or Load-AcquirePC instruction, such that the memory access generated by the Load-Acquire or Load-AcquirePC instruction is observed by each PE to the extent that the PE is required to observe the access coherently, before any of the memory accesses appearing in program order after the Load-Acquire or Load-AcquirePC instruction are observed by that PE to the extent that the PE is required to observe the accesses coherently.

The use of a Load-Acquire or Load-AcquirePC instruction creates order between the Memory effects of instructions as described in the definition of Barrier-ordered-before.

The full definition of the Store-Release instruction is covered formally in the Ordering requirements defined by the formal concurrency model and this introduction to the Store-Release instruction is not intended to contradict that section.

The basic principle of a Store-Release instruction is to introduce order between the following:

- A set of memory accesses, RWx, that are generated by the PE executing the Store-Release instruction and that appear in program order before the Store-Release instruction, together with those that originate from a different PE to the extent that the PE is required to observe them coherently, observed by the PE before executing the Store-release.
- The memory access generated by the Store-Release (Wrel), such that all of the memory accesses, RWx, are observed by each PE to the extent that the PE is required to observe those accesses coherently, before Wrel is observed by that PE to the extent that the PE is required to observe that access coherently.

The use of a Store-Release instruction creates order between the Memory effects of instructions as described in the definition of Barrier-ordered-before.

Where a Load-Acquire appears in program order after a Store-Release, the memory access generated by the Store-Release instruction is observed by each PE to the extent that PE is required to observe the access coherently, before the memory access generated by the Load-Acquire instruction is observed by that PE, to the extent that the PE is required to observe the access coherently. In addition, the use of a Load-Acquire, Load-AcquirePC or a Store-Release instruction on accesses to a Memory-mapped peripheral introduces order between the Memory effects of the instructions that access that peripheral, as described in the definition of Peripheral arrival order.

Load-Acquire, Load-AcquirePC and Store-Release, other than Load-Acquire Exclusive Pair and Store-Release-Exclusive Pair, access only a single data element.

Load-Acquire Exclusive Pair and Store-Release Exclusive Pair access two data elements.

A Store-Release Exclusive instruction has the release semantics only if the store is successful.

Note

- Each Load-Acquire Exclusive and Store-Release Exclusive instruction is essentially a variant of the equivalent Load-Exclusive or Store-Exclusive instruction. All usage restrictions and single-copy atomicity properties:
  - That apply to the Load-Exclusive instructions also apply to the Load-Acquire Exclusive instructions.
  - That apply to the Store-Exclusive instructions also apply to the Store-Release Exclusive instructions.
- The Load-Acquire, Load-AcquirePC, and Store-Release instructions can remove the requirement to use the explicit `DMB` instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B1.6 Caches and memory hierarchy

The implementation of a memory system depends heavily on the microarchitecture and therefore many details of the memory system are IMPLEMENTATION DEFINED. The Arm architecture defines the application level interface to the memory system, including a hierarchical memory system with multiple levels of cache. This section describes an application level view of this system. It contains the subsections:

- Introduction to caches.
- Memory hierarchy.
- Application level access to functionality related to caches
- Implication of caches for the application programmer.
- Prefetching into cache.

B1.6.1 Introduction to caches

A cache is a block of high-speed memory that contains a number of entries, each consisting of:

- Main memory address information, commonly known as a *tag*.
- The associated data.

Caches increase the average speed of a memory access. Caching takes account of two principles of locality:

Spatial locality

An access to one Location is likely to be followed by accesses to adjacent Locations. Examples of this principle are:

- Sequential instruction execution.
- Accessing a data structure.

Temporal locality

An access to an area of memory is likely to be repeated in a short time period. An example of this principle is the execution of a software loop.

To minimize the quantity of control information stored, the spatial locality property groups several locations together under the same tag. This logical block is commonly known as a *cache line*. When data is loaded into a cache, access times for subsequent loads and stores are reduced, resulting in overall performance benefits. An access to information already in a cache is known as a *cache hit*, and other accesses are called *cache misses*.

Normally, caches are self-managing, with the updates occurring automatically. Whenever the PE accesses a cacheable memory location, the cache is checked. If the access is a cache hit, the access occurs in the cache. Otherwise, the access is made to memory. Typically, when making this access, a cache location is allocated and the cache line loaded from memory. The Arm architecture permits different cache topologies and access policies, provided they comply with the memory coherency model described in this manual.

Caches introduce a number of potential problems, mainly because:

- Memory accesses can occur at times other than when the programmer would expect them.
- A data item can be held in multiple physical locations.

B1.6.2 Memory hierarchy

Typically memory close to a PE has very low latency, but is limited in size and expensive to implement. Further from the PE it is common to implement larger blocks of memory but these have increased latency. To optimize overall performance, an Armv8 memory system can include multiple levels of cache in a hierarchical memory system that exploits this trade-off between size and latency. Figure B1-3 shows an example of such a system in an Armv8-R system that supports virtual addressing.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

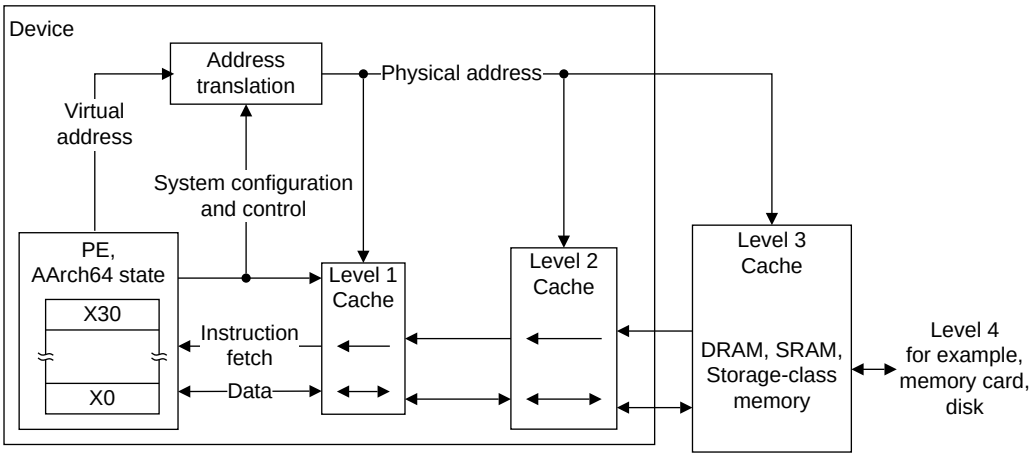


Figure B1-3 Multiple levels of cache in a memory hierarchy

Note

In this manual, in a hierarchical memory system, Level 1 refers to the level closest to the processing element, as shown in Figure B1-3.

Instructions and data can be held in separate caches or in a unified cache. A cache hierarchy can have one or more levels of separate instruction and data caches, with one or more unified caches that are located at the levels closest to the main memory. Memory coherency for cache topologies can be defined using the conceptual points Point of Unification (PoU), Point of Coherency (PoC), Point of Persistence (PoP), and Point of Deep Persistence (PoDP).

For more information, including the definitions of PoU, PoC, PoP, and PoDP, see About cache maintenance in AArch64 state.

B1.6.2.1 The cacheability and shareability memory attributes

Cacheability and shareability are two attributes that describe the memory hierarchy in a multiprocessing system:

Cacheability This attribute defines whether memory locations are allowed to be allocated into a cache or not. Cacheability is defined independently for Inner and Outer Cacheability locations.

Shareability This attribute defines whether memory locations are shareable between different agents in a system. Marking a memory location as shareable for a particular domain requires hardware to ensure that the location is coherent for all agents in that domain. Shareability is defined independently for Inner and Outer Shareability domains.

For more information about Cacheability and Shareability, see Memory types and attributes.

B1.6.3 Application level access to functionality related to caches

As indicated in About the Application level programmers' model, the application level corresponds to execution at EL0. The architecture defines a set of cache maintenance instructions that software can use to manage cache coherency. Software executing at a higher Exception level can enable use of some of this functionality from EL0, as follows:

When the value of SCTLR_EL1.UCI is 1

Software executing at EL0 can access:

- The data cache maintenance instructions, `DC CVAU`, `DC CVAC`, `DC CVAP`, `DC CVADP`, and `DC CIVAC`. See The data cache maintenance instruction (DC).
- The instruction cache maintenance instruction `IC IVAU`. See The instruction cache maintenance instruction (IC).

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Attempted execution of these instructions might generate a Permission fault as described in Permission fault.

When the value of SCTLR_EL1.UCT is 1

Software executing at EL0 can access the cache type register. See CTR_EL0.

When the value of SCTLR_EL1.DZE is 1

Software executing at EL0 can access the data cache zero instruction `DC ZVA`. See Data cache zero instruction.

The SCTLR_EL1.{UCI, UCT, DZE} control fields are accessible only by software executing at EL1 or higher.

When the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, the controls {UCI, UCT and DZE} are found in SCTLR_EL2.

This functionality is UNDEFINED at EL0 when the value of the corresponding SCTLR_EL1 control field is 0.

B1.6.4 Implication of caches for the application programmer

In normal operation, the caches are largely invisible to the application programmer. However they can become visible when there is a breakdown in the coherency of the caches. Such a breakdown can occur:

- When memory locations are updated by other agents in the system that do not use hardware management of coherency.
- When memory updates made from the application software must be made visible to other agents in the system, without the use of hardware management of coherency.

For example:

- In the absence of hardware management of coherency of DMA accesses, in a system with a DMA controller that reads memory locations that are held in the data cache of a PE, a breakdown of coherency occurs when the PE has written new data in the data cache, but the DMA controller reads the old data held in memory.
- In a Harvard cache implementation, where there are separate instruction and data caches, a breakdown of coherency occurs when new instruction data has been written into the data cache, but the instruction cache still contains the old instruction data.

B1.6.4.1 Data coherency issues

Software can ensure the data coherency of caches in the following ways:

- By not using the caches in situations where coherency issues can arise. This can be achieved by:
  - Using Non-cacheable or, in some cases, Write-Through Cacheable memory.
  - Not enabling caches in the system.
- By using cache maintenance instructions to manage the coherency issues in software. See Application level access to functionality related to caches.
- By using hardware coherency mechanisms to ensure the coherency of data accesses to memory for cacheable locations by observers within the different shareability domains, see Non-shareable Normal memory and Shareable, Inner Shareable, and Outer Shareable Normal memory.

Note

The performance of these hardware coherency mechanisms is highly IMPLEMENTATION SPECIFIC. In some implementations, the mechanism suppresses the ability to cache shareable locations. In other implementations, cache coherency hardware can hold data in caches while managing coherency between observers within the shareability domains.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Not all these mechanisms are directly available to software operating at EL0 and might involve interaction with software operating at a higher Exception level.

B1.6.4.2 Synchronization and coherency issues between data and instruction accesses

How far ahead of the current point of execution instructions are fetched from is IMPLEMENTATION DEFINED. Such prefetching can be either a fixed or a dynamically varying number of instructions, and can follow any or all possible future execution paths. For all types of memory:

- The PE might have fetched the instructions from memory at any time since the last Context Synchronization event on that PE.
- Any instructions fetched in this way might be executed multiple times, if this is required by the execution of the program, without being refetched from memory. In the absence of a Context Synchronization event, there is no limit on the number of times such an instruction might be executed without being refetched from memory.

The Arm architecture requires the hardware to ensure coherency between instruction caches and memory, even for locations of shared memory. A write has been made coherent with an instruction fetch of a shareability domain when:

CTR_EL0.{DIC, IDC} == {0, 0} The location written to has been cleaned to the Point of unification (PoU) from the data cache, and that clean is complete for the shareability domain. Subsequently the location has been invalidated to the Point of unification (PoU) from the instruction cache, and that invalidation is complete for the shareability domain.

CTR_EL0.{DIC, IDC} == {0, 1} The write is complete for the shareability domain. Subsequently the location has been invalidated to the Point of unification (PoU) from the instruction cache, and that invalidation is complete for the shareability domain.

CTR_EL0.{DIC, IDC} == {1, 1} The write is complete for the shareability domain.

Note

Microarchitecturally, this means that these situations cannot both be true in an implementation:

- After delays in fetching from memory, the instruction queue can have entries written into it out of order.
- For an implementation:
  - When CTR_EL0.DIC == 0, if there is an outstanding entry in the instruction queue, then later entries in the instruction queue are not impacted by the IC IVAU instructions of a different core.
  - When CTR_EL0.DIC == 1, if there is a write to the location that is held in the queue when there is an outstanding entry in the instruction queue for an older entry, then the instruction queue does not have entries invalidated from it.

The full definition of the ordering implications of CTR_EL0.{DIC, IDC} is covered formally in the Ordering requirements defined by the formal concurrency model and the information in this section is not intended to contradict that section.

If software requires coherency between instruction execution and memory, it must manage this coherency using Context Synchronization events and cache maintenance instructions. The following code sequence can be used to allow a PE to execute code that the same PE has written.

```
; Coherency example for data and instruction accesses within the same Inner Shareable domain.
; Enter this code with <Wt> containing a new 32-bit instruction,
; to be held in Cacheable space at a location pointed to by Xn.
        STR Wt, [Xn]
        DC CVAU, Xn         ; Clean data cache by VA to point of unification (PoU)
        DSB ISH             ; Ensure visibility of the data cleaned from cache
        IC IVAU, Xn         ; Invalidate instruction cache by VA to PoU
        DSB ISH             ; Ensure completion of the invalidations
        ISB                 ; Synchronize the fetched instruction stream
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

- If this sequence is not executed between writing data to a location and executing the instruction at that location, the lack of coherency between instruction caches and memory means that the instructions that are executed might be the old instruction or the updated instruction, and which is used can arbitrarily vary during execution. It must not be assumed by software, before the synchronization sequence is executed, that when the updated instruction has been seen, the old instruction will not be seen again.
- For Non-cacheable or Write-Through accesses, the clean data cache by VA instruction is not required. However, the invalidate instruction cache instruction is required because the Armv8 AArch64 architecture allows Non-cacheable accesses to be held in an instruction cache. See Non-cacheable accesses and instruction caches.
- This code can be used when the thread of execution modifying the code is the same thread of execution that is executing the code. The Arm architecture limits the set of instructions that can be executed by one thread of execution as they are being modified by another thread of execution without requiring explicit synchronization. See Concurrent modification and execution of instructions.
- The system software controls whether these cache maintenance instructions are available to the application level by setting SCTLR_EL1.UCI.

B1.6.5 Prefetching into cache

The Arm architecture provides memory system hints `PRFM`, `PRFUM`, `LDNP`, and `STNP` that software can use to communicate the expected use of memory locations to the hardware. The memory system can respond by taking actions that are expected to speed up the memory accesses if they occur. The effect of these memory system hints is IMPLEMENTATION DEFINED. Typically, implementations use this information to bring the data or instruction locations into caches.

The Prefetch instructions are hints, and so implementations can treat them as `NOP`s without affecting the functional behavior of the device. The instructions cannot generate synchronous Data Abort exceptions, but the resulting memory system operations might, under exceptional circumstances, generate an asynchronous External abort, which is taken using an SError exception. For more information, see ISS encoding for an exception from a Data Abort.

`PrefetchHint {}` defines the prefetch hint types.

The `Hint_Prefetch()` function signals to the memory system that memory accesses of the type `hint` to or from the specified `address` are likely to occur in the near future. The memory system might take some action to speed up the memory accesses when they do occur, such as prefetching the specified address into one or more caches as indicated by the innermost cache level `target` and non-temporal hint `stream`.

For more information on `PRFM`, `PRFUM` and load/store instructions that provide hints to the memory system, see Prefetch memory and Load/store SIMD and floating-point non-temporal pair.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B1.7 Alignment support

This section describes alignment support. It contains the following subsections:

- Instruction alignment.
- Alignment of data accesses.

B1.7.1 Instruction alignment

A64 instructions must be word-aligned.

Attempting to fetch an instruction from a misaligned location results in a PC alignment fault. See PC alignment checking.

B1.7.2 Alignment of data accesses

When instructions that load or store single or multiple registers are executed, if the value of SCTLR_ELx.A applicable to the current Exception level is 1, an Alignment fault is generated if any of the following apply:

- For Load-Exclusive, Store-Exclusive, and Atomic instructions, the address that is accessed is not aligned to the overall access size.
- For all other instructions, the address that is accessed is not aligned to the size of the data element being accessed.

If FEAT_LSE2 is implemented, Load-Acquire/Store-Release instructions without exclusive or atomic behavior generate an Alignment fault if all of the following apply:

- Not all bytes of the memory access lie within a 16-byte quantity aligned to 16 bytes.
- The value of SCTLR_ELx.nAA applicable to the current Exception level is 0.

For an unaligned access to any type of Device memory:

- If the memory location cannot support unaligned accesses then an Alignment fault is generated.
- If the memory location supports unaligned accesses, then it is IMPLEMENTATION DEFINED whether the access generates an Alignment fault if the location would not generate an Alignment fault if the same access were made to Normal memory.

B1.7.2.1 Unaligned accesses to Normal memory

The behavior of unaligned accesses to Normal memory is dependent on all of the following:

- The instruction causing the memory access.
- The memory attributes of the accessed memory.
- The value of SCTLR_ELx.{A, nAA}.
- Whether or not FEAT_LSE2 is implemented.

B1.7.2.1.1 Load or Store of Single or Multiple registers

For all instructions that load or store single or multiple registers, but not Load-Exclusive, Store-Exclusive, Load-Acquire/Store-Release, and Atomic instructions, if the address that is accessed is not aligned to the size of the data element being accessed, then:

When the value of SCTLR_ELx.A applicable to the current Exception level is 1, an Alignment fault is generated.

When the value of SCTLR_ELx.A applicable to the current Exception level is 0:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- An unaligned access is performed.
- If FEAT_LSE2 is not implemented, the access is not guaranteed to be single-copy atomic except at the byte access level.
- If FEAT_LSE2 is implemented:
  - If all the bytes of the memory access lie within a 16-byte quantity aligned to 16 bytes and are to Normal Inner Write-Back, Outer Write-Back Cacheable, Shareable memory, the memory access is single-copy atomic. For `LDNP`, `LDP`, or `STP` instructions, the entire memory access will be single-copy atomic.
  - If all the bytes of the memory accessed do not lie within a 16-byte quantity aligned to 16 bytes, or the access is not to Normal Inner Write-Back, Outer Write-Back Cacheable, Shareable memory, the access is not guaranteed to be single-copy atomic except at the byte access level.

For these instructions, the definition of an unaligned access is based on the size of the accessed elements, not the overall size of the memory access. This affects SIMD loads and stores, and also load/store pair instructions.

B1.7.2.1.2 Load-Exclusive/ Store-Exclusive and Atomic instructions

If FEAT_LSE2 is not implemented and the value of SCTLR_ELx.A applicable to the current Exception level is 0, Load-Exclusive/Store-Exclusive and Atomic instructions, including those with acquire or release semantics, generate an alignment fault when the address being accessed is not aligned to the overall size of the access.

If FEAT_LSE2 is implemented, then:

- If all the bytes of the memory access lie within a 16-byte quantity aligned to 16 bytes and are to Normal Inner Write-Back, Outer Write-Back Cacheable, Shareable memory, an unaligned access is performed.
- If all the bytes of the memory access do not lie within a 16-byte quantity aligned to 16-bytes, or the memory access is not to Normal Inner Write-Back, Outer Write-Back Cacheable, Shareable memory, then it is a CONSTRAINED UNPREDICTABLE choice of either of the following:
  - An unaligned access is performed meeting all of the semantics of the instruction.
  - An Alignment fault is generated.

Where memory access is performed, then it is single-copy atomic.

For these instructions, the definition of an unaligned access is based on the overall access size.

If FEAT_LS64 is implemented, when a single-copy atomic 64-byte instruction accesses a memory location that is not aligned to 64 bytes, an Alignment fault always occurs, regardless of the value of SCTLR_ELx.A.

B1.7.2.1.3 Non-atomic Load-Acquire/Store-Release instructions

For Load-Acquire/Store-Release instructions that do not have exclusive or atomic behaviors, when the address being accessed is not aligned to the size of the data element being accessed and the value of SCTLR_ELx.A applicable to the current Exception level is 0:

If FEAT_LSE2 is not implemented, then these instructions generate an Alignment fault.

If FEAT_LSE2 is implemented, then:

- If the memory access is not to Normal Inner Write-Back, Outer Write-Back Cacheable, Shareable memory, then it is a CONSTRAINED UNPREDICTABLE choice of either of the following:
  - An unaligned access is performed which is not guaranteed to be single-copy atomic except at the byte access level.
  - An Alignment fault is generated.
- If all of the bytes of the memory access do not lie within a 16-byte quantity aligned to 16 bytes and SCTLR_ELx.nAA applicable to the current Exception level is 1, then an unaligned access is performed which is not guaranteed to be single-copy atomic except at the byte access level.

In this case, when an access is performed it meets all the semantics of the instruction, but the architecture does not define the order of the different transactions of the access defined by the single instructions relative to each other.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If all the bytes of the memory access lie within a 16-byte quantity aligned to 16 bytes and are to Normal Inner Write-Back, Outer Write-Back Cacheable, Shareable memory, an unaligned access meeting all the semantics of the instruction is performed.

Note

- Unaligned accesses typically take additional cycles to complete compared to a naturally-aligned access.
- An operation that is not single-copy atomic above the byte level can abort on any memory access that it makes and can abort on more than one access. This means that an unaligned access that occurs across a page boundary can generate an abort on either side of the page boundary.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B1.8 Endian support

General description of endianness in the Arm architecture describes the relationship between endianness and memory addressing in the Arm architecture.

The following subsections then describe the endianness schemes supported by the architecture:

- Instruction endianness.
- Data endianness.
- Endianness of memory-mapped peripherals.

B1.8.1 General description of endianness in the Arm architecture

This section describes only memory addressing and the effects of endianness for data elements up to quadwords of 128 bits. However, this description can be extended to apply to larger data elements.

For an address A, Figure B1-4 shows, for big-endian and little-endian memory systems, the relationship between:

- The quadword at address A.
- The doubleword at address A and A+8.
- The words at addresses A, A+4, A+8, and A+12.
- The halfwords at addresses A, A+2, A+4, A+6, A+8, A+10, A+12, and A+14.
- The bytes at addresses A, A+1, A+2, A+3, A+4, A+5, A+6, A+7, A+8, A+9, A+10, A+11, A+12, A+13, A+14, and A+15.

The terms in Figure B1-4 have the following definitions:

B_A Byte at address A.

HW_A Halfword at address A.

MSByte Most significant byte.

LSByte Least significant byte.

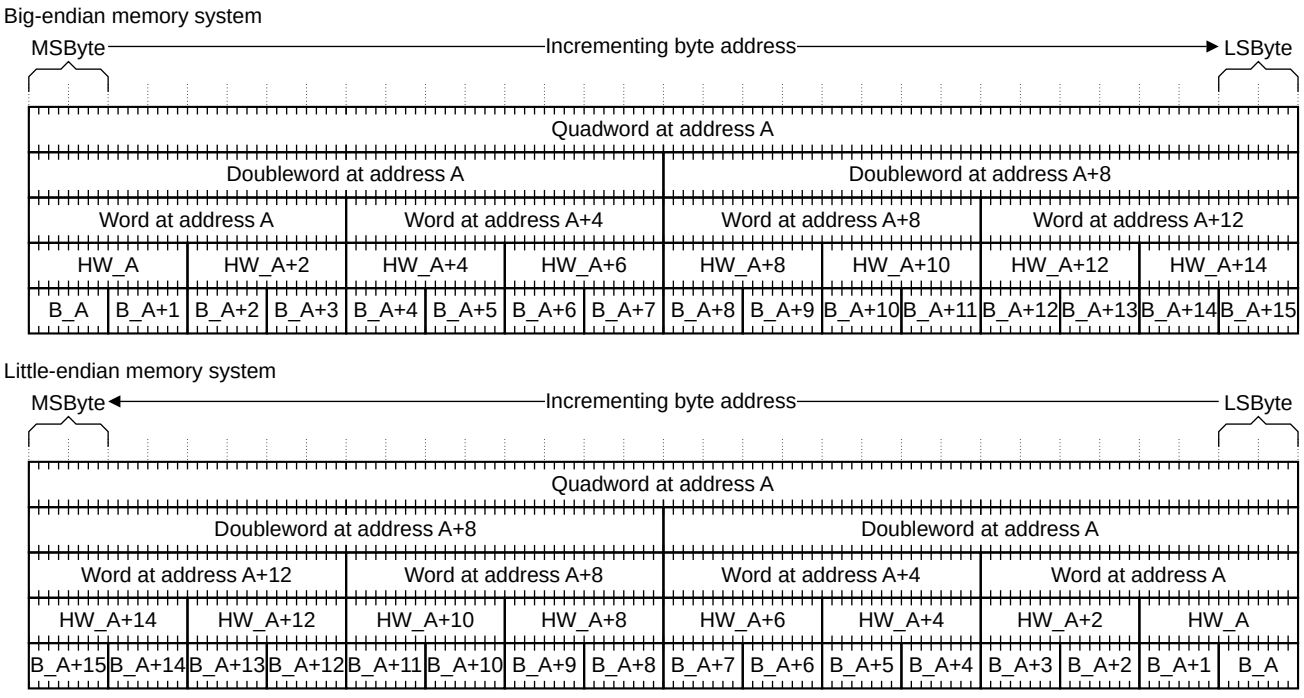


Figure B1-4 Endianness relationships

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The big-endian and little-endian mapping schemes determine the order in which the bytes of a quadword, doubleword, word, or halfword are interpreted. For example, a load of a word from address `0x1000` always results in an access to the bytes at memory locations `0x1000`, `0x1001`, `0x1002`, and `0x1003`. The endianness mapping scheme determines the significance of these 4 bytes.

B1.8.2 Instruction endianness

A64 instructions have a fixed length of 32 bits and are always little-endian.

B1.8.3 Data endianness

SCTLR_EL1.E0E, configurable at EL1 or higher, determines the data endianness for execution at EL0. When the Effective value of HCR_EL2.{E2H,TGE} is {1, 1}, the control is from SCTLR_EL2.E0E.

The data size used for endianness conversions:

- Is the size of the data value that is loaded or stored for SIMD and floating-point register and general-purpose register loads and stores.
- Is the size of the data element that is loaded or stored for SIMD element and data structure loads and stores. For more information, see Endianness in SIMD operations.

Note

This means the Armv8 architecture introduces a requirement for 128-bit endian conversions.

B1.8.3.1 Instructions to reverse bytes in a general-purpose register, a SIMD and floating-point register

An application or device driver might have to interface to memory-mapped peripheral registers or shared memory structures that are not the same endianness as the internal data structures. Similarly, the endianness of the operating system might not match that of the peripheral registers or shared memory. In these cases, the PE requires an efficient method to transform explicitly the endianness of the data.

Table B1-4 shows the instructions that provide this functionality:

Table B1-4 Byte reversal instructions

Function	Instructions	Notes
Reverse bytes in 32-bit word or words[a]	REV32	For use with general-purpose registers
Reverse bytes in whole register	REV	For use with general-purpose registers
Reverse bytes in 16-bit halfwords	REV16	For use with general-purpose registers
Reverse elements in doublewords, vector	REV64	For use with SIMD and floating-point registers
Reverse elements in words, vector	REV32	For use with SIMD and floating-point registers
Reverse elements in halfwords, vector	REV16	For use with SIMD and floating-point registers

a. Can operate on multiple words.

B1.8.3.2 Endianness in SIMD operations

SIMD element load/store instructions transfer vectors of elements between memory and the SIMD and floating-point register file. An instruction specifies both the length of the transfer and the size of the data elements being transferred. This information is used to load and store data correctly in both big-endian and little-endian systems.

For example:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
LD1 {V0.4H}, [X1]
```

This loads a 64-bit register with four 16-bit values. The four elements appear in the register in array order, with the lowest indexed element fetched from the lowest address. The order of bytes in the elements depends on the endianness configuration, as shown in Figure B1-5. Therefore, the order of the elements in the registers is the same regardless of the endianness configuration.

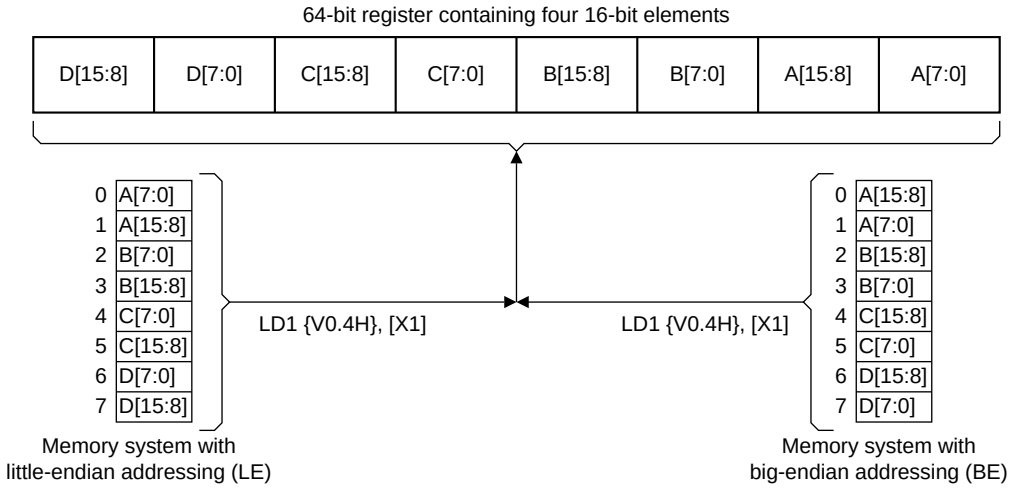


Figure B1-5 SIMD byte order example

The `BigEndian()` pseudocode function determines the current endianness of the data.

The `BigEndianReverse()` pseudocode function reverses the endianness of a bitstring.

The `BigEndian()` and `BigEndianReverse()` functions are defined in A-profile Architecture Pseudocode.

B1.8.4 Endianness of memory-mapped peripherals

All memory-mapped peripherals defined in the Arm architecture must be little-endian.

Peripherals to which this requirement applies include:

- Memory-mapped register interfaces to a debugger, or to a Cross Trigger Interface, see About the External Debug Registers.
- The memory-mapped register interface to the system level implementation of the Generic Timer, see System Level Implementation of the Generic Timer.
- A memory-mapped register interface to the Performance Monitors, see Recommended External Interface to the Performance Monitors.
- Memory-mapped register interfaces to an Arm Generic Interface Controller, see the *ARM® Generic Interrupt Controller Architecture Specification, GIC architecture version 3.0 and version 4.0.*
- The memory-mapped register interface to an Arm trace component. See, for example, the *Arm® Embedded Trace Macrocell Architecture Specification, ETMv4.*

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B1.9 Memory types and attributes

The ordering of accesses for addresses in memory, referred to as the memory order model, is defined by the memory attributes. The following sections describe this model:

- Normal memory.
- Device memory.
- Memory access restrictions.

B1.9.1 Normal memory

The Normal memory type attribute applies to most memory in a system. It indicates that the hardware is permitted by the architecture to perform Speculative data read accesses to these locations, regardless of the access permissions for these locations.

The Normal memory type has the following properties:

- An Explicit Memory Write Effect to a memory location with the Normal attribute completes in finite time.
- Explicit Memory Write Effects to a memory location with the Normal memory type that is either Non-cacheable or Write-Through cacheable for both the Inner and Outer cacheability must reach the endpoint for that location in the memory system in finite time. Two writes to the same location, where at least one is using the Normal memory type, might be merged before they reach the endpoint unless there is an ordered-before relationship between the two writes. For the purposes of this requirement, the endpoint for a location in Conventional memory is the PoC.
- Unaligned memory accesses can access Normal memory if the system is configured to generate such accesses.
- There is no requirement for the memory system beyond the PE to be able to identify the elements accessed by multi-register load/store instructions. See Multi-register loads and stores that access Normal memory.
- Where a load or store instruction performs a sequence of memory accesses, as opposed to one single-copy atomic access as defined in the rules for single-copy atomicity, these accesses might occur multiple times as a result of executing the load or store instruction.

Note

Write speculation that is visible to other observers is prohibited for all memory types.

Note

- The Normal memory attribute is appropriate for locations of memory that are idempotent, meaning that they exhibit all of the following properties:
  - Read accesses can be repeated with no side effects.
  - Repeated read accesses return the last value written to the resource being read.
  - Read accesses can fetch additional memory locations with no side-effects.
  - Write accesses can be repeated with no side-effects if the contents of the location accessed are unchanged between the repeated writes or as the result of an exception, as described in this section.
  - Unaligned accesses can be supported.
  - Accesses can be merged before accessing the target memory system.
- Normal memory allows speculative reads and may be affected by intermediate buffering and forwarding of data. If non-idempotent memory locations are mapped as Normal memory, the following may occur:
  - Memory accesses return UNKNOWN values.
  - UNPREDICTABLE effects on memory-mapped peripherals.
- An instruction that generates a sequence of accesses as described in Atomicity in the Arm architecture might be abandoned as a result of an exception being taken during the sequence of accesses. On return from the exception the instruction is restarted, and therefore, one or more of the memory locations might be accessed multiple times. This can result in repeated write accesses to a location that has been changed between the write accesses.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For accesses to Normal memory, a DMB instruction is required to ensure the required ordering.

The following sections describe the other attributes for Normal memory:

- Shareable Normal memory.
- Non-shareable Normal memory.
- Cacheability attributes for Normal memory.

See also:

- Multi-register loads and stores that access Normal memory.
- Atomicity in the Arm architecture.
- Memory barriers.
- Concurrent modification and execution of instructions.

B1.9.1.1 Shareable Normal memory

A Normal memory location has a Shareability attribute that is one of:

- Inner Shareable, meaning it applies across the Inner Shareable shareability domain.
- Outer Shareable, meaning it applies across both the Inner Shareable and the Outer Shareable shareability domains.
- Non-shareable.

The shareability attributes define the data coherency requirements of the location, which hardware must enforce. They do not affect the coherency requirements of instruction fetches, see Synchronization and coherency issues between data and instruction accesses.

Note

- System designers can use the shareability attribute to specify the locations in Normal memory for which coherency must be maintained. However, software developers must not assume that specifying a memory location as Non-shareable permits software to make assumptions about the incoherency of the location between different PEs in a shared memory system. Such assumptions are not portable between different multiprocessing implementations that might use the shareability attribute. Any multiprocessing implementation might implement caches that are shared, inherently, between different processing elements.
- This architecture assumes that all PEs that use the same operating system or hypervisor are in the same Inner Shareable shareability domain.

B1.9.1.1.1 Shareable, Inner Shareable, and Outer Shareable Normal memory

The Arm architecture abstracts the system as a series of Inner and Outer Shareability domains.

Each Inner Shareability domain contains a set of observers that are data coherent for each member of that set for data accesses with the Inner Shareable attribute made by any member of that set.

Each Outer Shareability domain contains a set of observers that are data coherent for each member of that set for data accesses with the Outer Shareable attribute made by any member of that set.

The following properties also hold:

- Each observer is a member of only a single Inner Shareability domain.
- Each observer is a member of only a single Outer Shareability domain.
- All observers in an Inner Shareability domain are always members of the same Outer Shareability domain. This means that an Inner Shareability domain is a subset of an Outer Shareability domain, although it is not required to be a proper subset.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

- Because all data accesses to Non-cacheable locations are data coherent to all observers, Non-cacheable locations are always treated as Outer Shareable.
- The Inner Shareable domain is expected to be the set of PEs controlled by a single hypervisor or operating system.

The details of the use of the shareability attributes are system-specific. Example B1-1 shows how they might be used.

Example B1-1 Use of shareability attributes

In an implementation, a particular subsystem with two clusters of PEs has the requirement that:

- In each cluster, the data caches or unified caches of the PEs in the cluster are transparent for all data accesses to memory locations with the Inner Shareable attribute.
- However, between the two clusters, the caches:
  - Are not required to be coherent for data accesses that have only the Inner Shareable attribute.
  - Are coherent for data accesses that have the Outer Shareable attribute.

In this system, each cluster is in a different shareability domain for the Inner Shareable attribute, but all components of the subsystem are in the same shareability domain for the Outer Shareable attribute.
A system might implement two such subsystems. If the data caches or unified caches of one subsystem are not transparent to the accesses from the other subsystem, this system has two Outer Shareable shareability domains.

Having two levels of shareability means system designers can reduce the performance and power overhead for shared memory locations that do not need to be part of the Outer Shareable shareability domain.

For shareable Normal memory, the Load-Exclusive and Store-Exclusive synchronization primitives take account of the possibility of accesses by more than one observer in the same Shareability domain.

B1.9.1.2 Non-shareable Normal memory

For Normal memory locations, the Non-shareable attribute identifies Normal memory that is likely to be accessed only by a single PE.

A location in Normal memory with the Non-shareable attribute does not require the hardware to make data accesses by different observers coherent, unless the memory is Non-cacheable. For a Non-shareable location, if other observers share the memory system, software must use cache maintenance instructions, if the presence of caches might lead to coherency issues when communicating between the observers. This cache maintenance requirement is in addition to the barrier operations that are required to ensure memory ordering.

For Non-shareable Normal memory, it is IMPLEMENTATION DEFINED whether the Load-Exclusive and Store-Exclusive synchronization primitives take account of the possibility of accesses by more than one observer.

B1.9.1.3 Cacheability attributes for Normal memory

In addition to being Outer Shareable, Inner Shareable or Non-shareable, each region of Normal memory is assigned a Cacheability attribute that is one of:

- Write-Through Cacheable.
- Write-Back Cacheable.
- Non-cacheable.

Also, for Write-Through Cacheable and Write-Back Cacheable Normal memory regions:

- A region might be assigned cache allocation hints for read and write accesses.
- It is IMPLEMENTATION DEFINED whether the cache allocation hints can have an additional attribute of Transient or Non-transient.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For more information, see Cacheability, cache allocation hints, and cache transient hints.

A memory location can be marked as having different cacheability attributes, for example when using aliases in a VA to PA mapping:

- If the attributes differ only in the cache allocation hint, this does not affect the behavior of accesses to that location.
- For other cases, see Mismatched memory attributes.

The cacheability attributes provide a mechanism of coherency control with observers that lie outside the shareability domain of a region of memory. In some cases, the use of Write-Through Cacheable or Non-cacheable regions of memory might provide a better mechanism for controlling coherency than the use of hardware coherency mechanisms or the use of cache maintenance routines. To this end, the architecture requires the following properties for Non-cacheable or Write-Through Cacheable memory:

- A completed write to a memory location that is Non-cacheable or Write-Through Cacheable for a level of cache made by an observer accessing the memory system inside the level of cache is visible to all observers accessing the memory system outside the level of cache without the need of explicit cache maintenance.
- A completed write to a memory location that is Non-cacheable for a level of cache made by an observer accessing the memory system outside the level of cache is visible to all observers accessing the memory system inside the level of cache without the need of explicit cache maintenance.
- For accesses to Normal memory that is Non-cacheable, a DMB instruction introduces a Barrier-ordered-before relation on all accesses to a single peripheral or block of memory that is of IMPLEMENTATION DEFINED size. For more information, see Ordering relations.

Note

Implementations can use the cache allocation hints to indicate a probable performance benefit of caching. For example, a programmer might know that a piece of memory is not going to be accessed again and would be better treated as Non-cacheable. The distinction between memory regions with attributes that differ only in the cache allocation hints exists only as a hint for performance.

For Normal memory, the Arm architecture provides cacheability attributes that are defined independently for each of two conceptual levels of cache, the *inner* and the *outer* cache. The relationship between these conceptual levels of cache and the implemented physical levels of cache is IMPLEMENTATION DEFINED, and can differ from the boundaries between the Inner and Outer Shareability domains. However:

- Inner refers to the innermost caches, meaning the caches that are closest to the PE, and always includes the lowest level of cache.
- No cache that is controlled by the Inner cacheability attributes can lie outside a cache that is controlled by the Outer cacheability attributes.
- An implementation might not have any outer cache.

Example B1-2, Example B1-3, and Example B1-4 describe the possible ways of implementing a system with three levels of cache, *level 1* (L1) to *level 3* (L3).

Note

- L1 cache is the level closest to the PE, see Memory hierarchy.
- When managing coherency, system designs must consider both the inner and outer cacheability attributes, as well as the shareability attributes. This is because hardware might have to manage the coherency of caches at one conceptual level, even when another conceptual level has the Non-cacheable attribute.

Example B1-2 Implementation with two inner and one outer cache levels

Implement the three levels of cache in the system, L1 to L3, with:

- The Inner cacheability attribute applied to L1 and L2 cache.
- The Outer cacheability attribute applied to L3 cache.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example B1-3 Implementation with three inner and no outer cache levels

Implement the three levels of cache in the system, L1 to L3, with the Inner cacheability attribute applied to L1, L2, and L3 cache. Do not use the Outer cacheability attribute.

Example B1-4 Implementation with one inner and two outer cache levels

Implement the three levels of cache in the system, L1 to L3, with:

- The Inner cacheability attribute applied to L1 cache.
- The Outer cacheability attribute applied to L2 and L3 cache.

B1.9.1.4 Multi-register loads and stores that access Normal memory

For all instructions that load or store more than one general-purpose register from an Exception level there is no requirement for the memory system beyond the PE to be able to identify the size of the elements accessed by these load or store instructions.

For all instructions that load or store more than one general-purpose register from an Exception level the order in which the registers are accessed is not defined by the architecture.

For all instructions that load or store one or more Advanced SIMD&FP registers from an Exception level, there is no requirement for the memory system beyond the PE to be able to identify the size of the element accessed by these load or store instructions.

B1.9.2 Device memory

The Device memory type attributes define memory locations where an access to the location can cause side-effects, or where the value returned for a load can vary depending on the number of loads performed. Typically, the Device memory attributes are used for memory-mapped peripherals and similar locations.

The attributes for Armv8 Device memory are:

Gathering Identified as G or nG, see Gathering.

Reordering Identified as R or nR, see Reordering.

Early Write Acknowledgement

Identified as E or nE, see Early Write Acknowledgement.

The Armv8 Device memory types are:

Device-nGnRnE Device non-Gathering, non-Reordering, No Early Write Acknowledgement.

Equivalent to the Strongly-ordered memory type in earlier versions of the architecture.

Device-nGnRE Device non-Gathering, non-Reordering, Early Write Acknowledgement.

Equivalent to the Device memory type in earlier versions of the architecture.

Device-nGRE Device non-Gathering, Reordering, Early Write Acknowledgement.

Armv8 adds this memory type to the translation table formats found in earlier versions of the architecture. The use of barriers is required to order accesses to Device-nGRE memory.

Device-GRE Device Gathering, Reordering, Early Write Acknowledgement.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Armv8 adds this memory type to the translation table formats found in earlier versions of the architecture. Device-GRE memory has the fewest constraints. It behaves similar to Normal memory, with the restriction that Speculative accesses to Device-GRE memory is forbidden.

Collectively these are referred to as *any Device memory type*. Going down the list, the memory types are described as getting *weaker*; conversely the going up the list the memory types are described as getting *stronger*.

Note

- As the list of types shows, these additional attributes are hierarchical. For example, a memory location that permits Gathering must also permit Reordering and Early Write Acknowledgement.
- The architecture does not require an implementation to distinguish between each of these memory types and Arm recognizes that not all implementations will do so. The subsection that describes each of the attributes, describes the implementation rules for the attribute.

All of these memory types have the following properties:

- Speculative data accesses are not permitted to any memory location with any Device memory attribute. This means that each memory access to any Device memory type must be one that would be generated by a simple sequential execution of the program.

  The following exception to this applies:

  — Reads generated by the SIMD and floating-point instructions can access bytes that are not explicitly accessed by the instruction if the bytes accessed are in a 16-byte window, aligned to 16-bytes, that contains at least one byte that is explicitly accessed by the instruction.

  — For Device memory with the Gathering attribute, reads generated by the `LDNP` instructions are permitted to access bytes that are not explicitly accessed by the instruction, provided that the bytes accessed are in a 128-byte window, aligned to 128-bytes, that contains at least one byte that is explicitly accessed by the instruction.

  — Where a load or store instruction performs a sequence of memory accesses, as opposed to one single-copy atomic access as defined in the rules for single-copy atomicity, these accesses might occur multiple times as a result of executing the load or store instruction. See Properties of single-copy atomic accesses.

  — An `LDRAA` or `LDRAB` instruction that fails the pointer authentication check and loads from a location in Device memory is permitted to cause one read access to that location if all of the other requirements for accessing that Device location are met.

  Note

  — An instruction that generates a sequence of accesses as described in Atomicity in the Arm architecture might be abandoned as a result of an exception being taken during the sequence of accesses. On return from the exception, the instruction is restarted, and therefore, one or more of the memory locations might be accessed multiple times. This can result in repeated accesses to a location where the program defines only a single access. For this reason, Arm strongly recommends that no accesses to Device memory are performed from a single instruction that spans the boundary of a translation granule or which in some other way could lead to some of the accesses being aborted.

  — Write speculation that is visible to other observers is prohibited for all memory types.

- An Explicit Memory Write Effect to a memory location with any Device memory type completes in finite time.
- If a value that would be returned from a read of a memory location with the Device memory type changes without an explicit Memory Write effect by an observer, this change must also be globally observed for all observers in the system in finite time. Such a change might occur in a peripheral location that holds status information.
- Data accesses to memory locations are coherent for all observers in the system, and correspondingly are treated as being Outer Shareable.
- A memory location with any Device memory attribute cannot be allocated into a cache.
- Explicit Memory Write Effects to a memory location with any Device memory attribute must reach the endpoint for that address in the memory system in finite time. Two Explicit Memory Write Effects of Device memory type to the same location might be merged before they reach the endpoint, unless both Explicit Memory Write Effects have the non-Gathering attribute or there is an ordered-before relationship between the two Explicit Memory Write Effects.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- For accesses to any Device memory type, a DMB instruction introduces a Barrier-ordered-before relation on all accesses to a single peripheral or block of memory that is of IMPLEMENTATION DEFINED size. For more information, see Ordering relations.
- If a memory location is not capable of supporting unaligned memory accesses, then an unaligned access to that memory location generates an Alignment fault at the first stage of translation that defined the location as being Device.
- If a memory location is capable of supporting unaligned memory accesses, and such a memory location is marked as Device, then it is IMPLEMENTATION DEFINED whether an unaligned access to that memory location generates an Alignment fault at the first stage of translation that defined the location as being Device.
- Hardware does not prevent speculative instruction fetches from a memory location with any of the Device memory attributes unless the memory location is also marked as execute-never for all Exception levels.

  Note

  This means that to prevent speculative instruction fetches from memory locations with Device memory attributes, any location that is assigned any Device memory type must also be marked as execute-never for all Exception levels. Failure to mark a memory location with any Device memory attribute as execute-never for all Exception levels is a programming error.

Note

In the EL1&0 translation regime in systems where HCR_EL2.TGE == 1 and HCR_EL2.DC == 0, any Alignment fault that results from the fact that all locations are treated as Device is a fault at the first stage of translation. This causes ESR_EL2.ISS[24] to be 0.

See also Memory access restrictions.

The memory types for translation table walks cannot be defined as any Device memory type within the TCR_ELx. For the EL1&0 translation regime, the memory accesses made during a stage 1 translation table walk are subject to a stage 2 translation, and as a result of this second stage of translation, the accesses from the first stage translation table walk might be made to memory locations with any Device memory type. These accesses might be made speculatively. When the value of the HCR_EL2.PTW bit is 1, a stage 2 Permission fault is generated if a first stage translation table walk is made to any Device memory type.

Note

In general, making a translation table walk to any Device memory type is the result of a programming error.

For an instruction fetch from a memory location with the Device attribute that is not marked as execute-never for the current Exception level, an implementation can either:

- Treat the instruction fetch as if it were to a memory location with the Normal Non-cacheable attribute.
- Take a Permission fault.

B1.9.2.1 Gathering

In the Device memory attribute:

G Indicates that the location has the Gathering attribute.

nG Indicates that the location does not have the Gathering attribute, meaning it is non-Gathering.

The Gathering attribute determines whether it is permissible for either:

- Multiple memory accesses of the same type, read or write, to the same memory location to be merged into a single transaction.
- Multiple memory accesses of the same type, read or write, to different memory locations to be merged into a single memory transaction on an interconnect.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

This also applies to writebacks from the cache, whether caused by a Natural eviction or as a result of a cache maintenance instruction.

For memory types with the Gathering attribute, either of these behaviors is permitted, provided that the ordering and coherency rules of the memory location are followed.

For memory types with the non-Gathering attribute, neither of these behaviors is permitted. As a result:

- The number of memory accesses that are made corresponds to the number that would be generated by a simple sequential execution of the program.
- All accesses occur at their single-copy atomic sizes, except that there is no requirement for the memory system beyond the PE to be able to identify the single-copy atomic sizes accessed by multi-register load/store instructions that generate more than one single-copy atomic access. See Multi-register loads and stores that access Device memory.

Gathering between two accesses, where one access is barrier-ordered-before the other access, is not permitted.

A read from a memory location with the non-Gathering attribute cannot come from a cache or a buffer, but must come from the endpoint for that address in the memory system. Typically this is a peripheral or physical memory.

Note

- A read from a memory location with the Gathering attribute can come from intermediate buffering of a previous write, provided that:
  - The accesses are not separated by a `DMB` or `DSB` barrier that affects both of the accesses.
  - The accesses are not separated by other ordering constructions that require that the accesses are in order. Such a construction might be a combination of Load-Acquire and Store-Release.
  - The accesses are not generated by a Store-Release instruction.
- The Arm architecture defines only programmer visible behavior. Therefore, gathering can be performed if a programmer cannot tell whether gathering has occurred.

An implementation is permitted to perform an access with the Gathering attribute in a manner consistent with the requirements specified by the non-Gathering attribute.

An implementation is not permitted to perform an access with the non-Gathering attribute in a manner consistent with the relaxations allowed by the Gathering attribute.

B1.9.2.2 Reordering

In the Device memory attribute:

R Indicates that the location has the Reordering attribute. Accesses to the location can be reordered within the same rules that apply to accesses to Normal Non-cacheable memory. All memory types with the Reordering attribute have the same ordering rules as accesses to Normal Non-cacheable memory, see Ordering relations.

nR Indicates that the location does not have the Reordering attribute, meaning it is non-Reordering.

Note

Some interconnect fabrics, such as PCIe, perform very limited reordering, which is not important for the software usage. It is outside the scope of the Arm architecture to prohibit the use of a non-Reordering memory type with these interconnects.

For all memory types with the non-Reordering attribute, the order of memory accesses arriving at a single peripheral of IMPLEMENTATION DEFINED size, as defined by the peripheral, must be the same order that occurs in a simple sequential execution of the program. That is, the accesses appear in program order. This ordering applies to all accesses using any of the memory types with the non-Reordering attribute. As a result, if there is a mixture of Device-nGnRE and Device-nGnRnE accesses to the same peripheral, these occur in program order. If the memory accesses are not to a peripheral, then this attribute imposes no restrictions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

- The IMPLEMENTATION DEFINED size of the single peripheral is the same as applies for the ordering guarantee provided by the DMB instruction.
- The Arm architecture defines only programmer visible behavior. Therefore, reordering can be performed if a programmer cannot tell whether reordering has occurred.
- The non-Reordering property is only required by the architecture to apply the order of arrival of accesses to a single memory-mapped peripheral of an IMPLEMENTATION DEFINED size, and is not required to have an impact on the order of observation of memory accesses to SDRAM. For this reason, there is no effect of the non-Reordering attribute on the ordering relations between accesses to different locations described in Ordering relations as part of the formal definition of the memory model. It does have an effect on the Peripheral arrival order described in Peripheral arrival order.
- If the same memory location is mapped with different aliases, and different attribute values, these are a type of mismatched attribute. The different attributes could be:
  - A different Reordering attribute value.
  - A different Device memory attribute value.

For information about the effects of accessing memory with mismatched attributes, see Mismatched memory attributes.

An implementation:

- Is permitted to perform an access with the Reordering attribute in a manner consistent with the requirements specified by the non-Reordering attribute.
- Is not permitted to perform an access with the non-Reordering attribute in a manner consistent with the relaxations allowed by the Reordering attribute.

The non-Reordering attribute does not require any additional ordering, other than that which applies to Normal memory, between:

- Accesses to one physical address with the non-Reordering attribute and accesses to a different physical address with the Reordering attribute.
- Access to one physical address with the non-Reordering attribute and access to a different physical address to Normal memory.
- Accesses with the non-Reordering attribute and accesses to different peripherals of IMPLEMENTATION DEFINED size.

The non-Reordering attribute has no effect on the ordering of cache maintenance instructions, even if the memory location specified in the instruction has the non-Reordering attribute.

B1.9.2.3 Early Write Acknowledgement

In the Device memory attribute:

E Indicates that the location has the Early Write Acknowledgement attribute.

nE Indicates that the location has the No Early Write Acknowledgement attribute.

If the No Early Write Acknowledgement attribute is assigned for a Device memory location:

- For memory system endpoints where the system architecture in which the PE is operating requires that acknowledgment of a write comes from the endpoint, it is guaranteed that:
  - Only the endpoint of the write access returns a write acknowledgment of the access.
  - No earlier point in the memory system returns a write acknowledgment.
- For memory system endpoints where the system architecture in which the PE is operating does not require that acknowledgment of a write comes from the endpoint, the acknowledgment of a write is not required to come from the endpoint.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

A write with the No Early Write Acknowledgement attribute assigned for a Device memory location is not expected to generate an abort in any situation where the equivalent write to the same location without the No Early Write Acknowledgement attribute assigned does not generate an abort.

This means that a `DSB` barrier instruction, executed by the PE that performed the write to the No Early Write Acknowledgement Location, completes only after the write has reached its endpoint in the memory system if that is required by the system architecture.

Peripherals are an example of system endpoints that require that the acknowledgment of a write comes from the endpoint.

Note

- The Early Write Acknowledgement attribute only affects where the endpoint acknowledgment is returned from, and does not affect the ordering of arrival at the endpoint between accesses, which is determined by either the Device Reordering attribute, or the use of barriers to create order.
- The areas of the physical memory map for which write acknowledgment from the endpoint is required is outside the scope of the Arm Architecture definition and must be defined as part of the system architecture in which the PE is operating. In particular, regions of memory handled as PCIe configuration writes are expected to support write acknowledgment from the endpoint.
- Arm recognizes that not all areas of a physical memory map will be capable of supporting write acknowledgment from the endpoint. In particular, Arm expects that regions of memory handled as posted writes under PCIe will not support write acknowledgment from the endpoint.
- For maximum software compatibility, Arm strongly recommends that all peripherals for which standard software drivers expect that the use of a `DSB` instruction will determine that a write has reached its endpoint are placed in areas of the physical memory map that support write acknowledgment from the endpoint.

B1.9.2.4 Multi-register loads and stores that access Device memory

For all instructions that load or store more than one general-purpose register and generate more than one single-copy atomic access for that load or store, there is no requirement for the memory system beyond the PE to be able to identify the single-copy atomic sizes accessed by these load or store instructions.

For all instructions that load or store more than one general-purpose register, the order in which the registers are accessed is not defined by the architecture. This applies even to accesses to any type of Device memory.

For all instructions that load or store one or more Advanced SIMD&FP, and generate more than one single-copy atomic access for that load or store, there is no requirement for the memory system beyond the PE to be able to identify the single-copy atomic sizes accessed by these load or store instructions, even for access to any type of Device memory.

The architecture permits that the non-speculative execution of an instruction that loads or stores more than one general-purpose or SIMD and floating-point register might result in repeated accesses to the same address, even if the resulting accesses are to any type of Device memory.

B1.9.3 Memory access restrictions

The following restrictions apply to memory accesses:

- For two explicit Memory Read effects to any two adjacent bytes in memory, p and $p+1$, generated by the same instruction, and for two explicit Memory Write effects to any two adjacent bytes in memory, p and $p+1$, that are generated by the same instruction:
  - The bytes p and $p+1$ must have the same memory type and Shareability attributes, otherwise the results are CONSTRAINED UNPREDICTABLE. For example, an `LD1`, `ST1`, or an unaligned load or store that spans the boundary between Normal memory and Device memory is CONSTRAINED UNPREDICTABLE.
  - Except for possible differences in the cache allocation hints, Arm deprecates having different cacheability attributes for bytes p and $p+1$.

  For the permitted CONSTRAINED UNPREDICTABLE behavior, see Crossing a page boundary with different memory types or Shareability attributes.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the accesses of an instruction that causes multiple accesses to any type of Device memory cross an address boundary that corresponds to the smallest implemented translation granule, then behavior is CONSTRAINED UNPREDICTABLE, and Crossing a peripheral boundary with a Device access describes the permitted behaviors. For this reason, it is important that an access to a volatile memory device is not made using a single instruction that crosses an address boundary of the size of the smallest implemented translation granule.

Note

— The boundary referred to is between two Device memory regions that are both of the size of the smallest implemented translation granule and aligned to the size of the smallest implemented translation granule.

— This restriction means it is important that an access to a volatile memory device is not made using a single instruction that crosses an address boundary of the size of the smallest implemented translation granule.

— Arm expects this restriction to constrain the placing of volatile memory devices in the system memory map, rather than expecting a compiler to be aware of the alignment of memory accesses.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B1.10 Mismatched memory attributes

Memory attributes are controlled by privileged software. For more information, see The AArch64 Virtual Memory System Architecture.

Physical memory locations are accessed with *mismatched attributes* if all accesses to the location do not use a common definition of all of the following attributes of that location:

- Memory type: Device-nGnRnE, Device-nGnRE, Device-nGRE, Device-GRE or Normal.
- Shareability.
- Cacheability, for the same level of the inner or outer cache, but excluding any cache allocation hints.

Collectively these are referred to as memory attributes.

Note

In this document, the terms *location* and *memory location* refer to any byte within the current coherency granule and are used interchangeably.

When a memory Location is accessed with mismatched attributes, the only software visible effects are one or more of the following:

- Uniprocessor semantics for reads and writes to that memory Location might be lost. This means:
  - A read of the memory Location by one agent might not return the value most recently written to that memory Location by the same agent.
  - Multiple writes to the memory Location by one agent with different memory attributes might not be ordered in program order.
- There might be a loss of coherency when multiple agents attempt to access a memory Location.
- There might be a loss of properties derived from the memory type, as described in later bullets in this section.
- If all Load-Exclusive/Store-Exclusive instructions executed across all threads to access a given memory Location do not use consistent memory attributes, the Exclusives monitor state becomes UNKNOWN.
- Bytes written without the Write-Back cacheable attribute within the same Write-Back granule as bytes written with the Write-Back cacheable attribute might have their values reverted to the old values as a result of cache Write-Back.

The loss of properties associated with mismatched memory type attributes refers only to the following properties of Device memory that are additional to the properties of Normal memory:

- Prohibition of Speculative read accesses.
- Prohibition on Gathering.
- Prohibition on reordering.

For the following situations, when a physical memory Location is accessed with mismatched attributes, a more restrictive set of behaviors applies. The description of each situation also describes the behaviors that apply:

1. Any agent that reads that memory Location using the same common definition of the Memory type, Shareability and Cacheability attributes is guaranteed to access it coherently, to the extent required by that common definition of the memory attributes, only if all the following conditions are met:
   - All writes are performed to an alias of the memory Location that uses the same definition of the Memory type, Shareability and Cacheability attributes.
   - All aliases with write permission have the Inner Cacheability attribute the same as the Outer Cacheability attribute.
   - Either:
     - All writes are performed to an alias of the memory Location that has Inner Cacheability and Outer Cacheability attributes both as Non-cacheable.
     - All aliases to a memory Location use a definition of the Shareability attributes that encompasses all the agents with permission to access the Location.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2. The possible software-visible effects caused by mismatched attributes for a memory Location are defined more precisely if all of the mismatched attributes define the memory Location as one of:
   - Any Device memory type.
   - Inner Non-cacheable, Outer Non-cacheable Normal memory.

   In these cases, the only permitted software-visible effects of the mismatched attributes are one or more of the following:
   - Possible loss of properties derived from the memory type when multiple agents attempt to access the memory Location.
   - Possible reordering of memory transactions to the same memory Location with different memory attributes, potentially leading to a loss of coherency or uniprocessor semantics. Any possible loss of coherency or uniprocessor semantics can be avoided by inserting DMB barrier instructions between accesses to the same memory Location that might use different attributes.

Where there is a loss of the uniprocessor semantics, ordering, or coherency, the following approaches can be used:

1. If the mismatched attributes for a memory location all assign the same shareability attribute to a Location that has a cacheable attribute, any loss of uniprocessor semantics, ordering, or coherency within a shareability domain can be avoided by use of software cache management. To do so, software must use the techniques that are required for the software management of the ordering or coherency of cacheable Locations between agents in different shareability domains. This means:
   - Before writing to a cacheable Location not using the Write-Back attribute, software must invalidate, or clean, a Location from the caches if any agent might have written to the Location with the Write-Back attribute. This avoids the possibility of overwriting the Location with stale data.
   - After writing to a cacheable Location with the Write-Back attribute, software must clean the Location from the caches, to make the write visible to external memory.
   - Before reading the Location with a cacheable attribute, software must invalidate, or clean and invalidate, the Location from the caches, to ensure that any value held in the caches reflects the last value made visible in external memory.
   - Executing a DMB barrier instruction, with scope that applies to the common shareability of the accesses, between any accesses to the same cacheable Location that use different attributes.

   In all cases:
   - Location refers to any byte within the current coherency granule.
   - A clean and invalidate instruction can be used instead of a clean instruction, or instead of an invalidate instruction.
   - In the sequences outlined in this section, all cache maintenance instructions and memory transactions must be completed, or ordered by the use of barrier operations, if they are not naturally ordered by the use of a common address, see Ordering and completion of data and instruction cache instructions.

   Note

   With software management of coherency, race conditions can cause loss of data. A race condition occurs when different agents write simultaneously to bytes that are in the same Location, and the invalidate, write, clean sequence of one agent overlaps with the equivalent sequence of another agent. A race condition also occurs if the first operation of either sequence is a clean, rather than an invalidate.

2. If the mismatched attributes for a Location mean that multiple cacheable accesses to the Location might be made with different shareability attributes, then uniprocessor semantics, ordering, and coherency are guaranteed only if:
   - Software running on a PE cleans and invalidates a Location from cache before and after each read or write to that Location by that PE.
   - A DMB barrier with scope that covers the full shareability of the accesses is placed between any accesses to the same memory Location that use different attributes.

   Note

   The Note in rule 1 of this list, about possible race conditions, also applies to this rule.

In addition, if multiple agents attempt to use Load-Exclusive or Store-Exclusive instructions to access a Location, and the accesses from the different agents have different memory attributes associated with the Location, the Exclusives monitor state becomes UNKNOWN.

Arm strongly recommends that software does not use mismatched attributes for aliases of the same Location. An implementation might not optimize the performance of a system that uses mismatched aliases.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

As described in Non-cacheable accesses and instruction caches, a non-cacheable access is permitted to be cached in an instruction cache, despite the fact that a non-cacheable access is not permitted to be cached in a unified cache. Despite this, when cacheable and non-cacheable aliases exist for memory which is executable, these must be treated as mismatched aliases to avoid coherency issues from the data or unified caches that might hold entries that will be brought into the instruction caches.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B1.11 Synchronization and semaphores

Armv8 provides non-blocking synchronization of shared memory, using *synchronization primitives*. The information in this section about memory accesses by synchronization primitives applies to accesses to both Normal memory and to any type of Device memory.

Note

Use of the Armv8 synchronization primitives scales for multiprocessing system designs.

Table B1-5 shows the synchronization primitives and the associated `CLREX` instruction.

Table B1-5 Synchronization primitives and associated instruction, A64 instruction set

Transaction size	Additional semantics	Load-Exclusive[a]	Store-Exclusive[a]	Other[a]
Byte	-	`LDXRB`	`STXRB`	-
	Load-Acquire/Store-Release	`LDAXRB`	`STLXRB`	-
Halfword	-	`LDXRH`	`STXRH`	-
	Load-Acquire/Store-Release	`LDAXRH`	`STLXRH`	-
Register[b]	-	`LDXR`	`STXR`	-
	Load-Acquire/Store-Release	`LDAXR`	`STLXR`	-
Pair[b]	-	`LDXP`	`STXP`	-
	Load-Acquire/Store-Release	`LDAXP`	`STLXP`	-
None	Clear-Exclusive	-	-	`CLREX`

a. Instruction in the A64 instruction set.

b. A register instruction operates on a doubleword if accessing an X register, or on a word if accessing a W register A pair instruction operates on two doublewords if access X registers, or on two words if accessing W registers.

Except for the row showing the `CLREX` instruction, the two instructions in a single row are a Load-Exclusive/Store-Exclusive instruction pair. The model for the use of a Load-Exclusive/Store-Exclusive instruction pair accessing a non-aborting memory address *x* is:

- The Load-Exclusive instruction reads a value from memory address *x*.
- The corresponding Store-Exclusive instruction succeeds in writing back to memory address *x* only if no other observer, process, or thread has performed a more recent store to address *x*. The Store-Exclusive instruction returns a status bit that indicates whether the Write Memory succeeded.

A Load-Exclusive instruction marks a small block of memory for exclusive access. The size of the marked block is IMPLEMENTATION DEFINED, see Marking and the size of the marked memory block. A Store-Exclusive instruction to any address in the marked block clears the marking.

Note

In this section, the term PE includes any observer that can generate a Load-Exclusive or a Store-Exclusive instruction.

The following sections give more information:

- Exclusive access instructions and Non-shareable memory locations.
- Exclusive access instructions and Shareable memory locations.
- Marking and the size of the marked memory block.
- Context switch support.
- Load-Exclusive and Store-Exclusive instruction usage restrictions.
- Use of WFE and SEV instructions by spin-locks.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B1.11.1 Exclusive access instructions and Non-shareable memory locations

For memory locations for which the shareability attribute is Non-shareable, the exclusive access instructions rely on a *local Exclusives monitor*, or *local monitor*, that marks the address of the Explicit Memory Read effect generated by any Load-Exclusive instruction executed by the PE.

A Load-Exclusive instruction performs a load from memory, and:

- The executing PE marks the physical memory address for exclusive access.
- The local monitor of the executing PE transitions to the Exclusive Access state.

A Store-Exclusive instruction performs a conditional store to memory that depends on the state of the local monitor:

If the local monitor is in the Exclusive Access state

- If the address of the Explicit Memory Write effect of the Store-Exclusive instruction is the same as the address that has been marked in the monitor due to the execution of an earlier Load-Exclusive instruction, then the store occurs. Otherwise, it is IMPLEMENTATION DEFINED whether the store occurs.
- A status value is returned to a register:
  - — If the store took place, the status value is 0.
  - — Otherwise, the status value is 1.
- The local monitor of the executing PE transitions to the Open Access state.

When an Exclusives monitor is in the Exclusive Access state, the monitor is *set.*

If the local monitor is in the Open Access state

- No store takes place.
- A status value of 1 is returned to a register.
- The local monitor remains in the Open Access state.

When an Exclusives monitor is in the Open Access state, the monitor is *clear*.

The Store-Exclusive instruction defines the register to which the status value is returned.

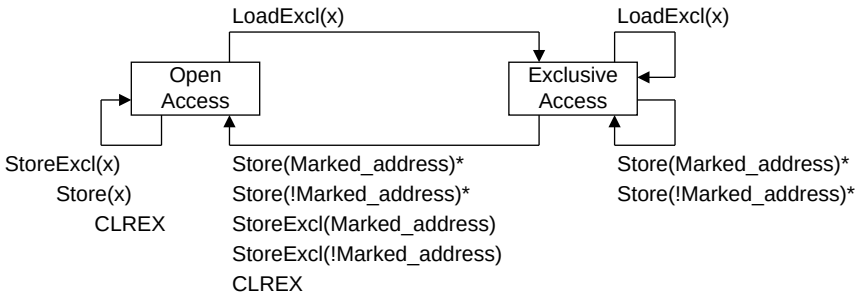
When a PE writes using any instruction other than a Store-Exclusive instruction:

- If the write is to a PA that is not marked as Exclusive Access by its local monitor and that local monitor is in the Exclusive Access state, it is IMPLEMENTATION DEFINED whether the write affects the state of the local monitor.
- If the write is to a PA that is marked as Exclusive Access by its local monitor, it is IMPLEMENTATION DEFINED whether the write affects the state of the local monitor.

It is IMPLEMENTATION DEFINED whether a store to a marked PA causes a mark in the local monitor to be cleared if that store is by an observer other than the one that caused the PA to be marked.

Figure B1-6 shows the state machine for the local monitor and the effect of each of the operations shown in the figure.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential



Operations marked * are possible alternative IMPLEMENTATION DEFINED options.
In the diagram: LoadExcl represents any Load-Exclusive instruction
StoreExcl represents any Store-Exclusive instruction
Store represents any other store instruction.

Any LoadExcl operation updates the marked address to the most significant bits of the address x used for the operation.

Figure B1-6 Local monitor state machine diagram

For more information about marking, see Marking and the size of the marked memory block.

Note

For the local monitor state machine, as shown in Figure B1-6:

- The IMPLEMENTATION DEFINED options for the local monitor are consistent with the local monitor being constructed so that it does not hold any PA, but instead treats any access as matching the address of the previous Load-Exclusive instruction.
- A local monitor implementation can be unaware of Load-Exclusive and Store-Exclusive instructions from other PEs.
- The architecture does not require a load instruction by another PE, that is not a Load-Exclusive instruction, to have any effect on the local monitor.
- It is IMPLEMENTATION DEFINED whether the transition from Exclusive Access to Open Access state occurs when the `Store` or `StoreExcl` is from another observer.

B1.11.1.1 Changes to the local monitor state resulting from speculative execution

The architecture permits a local monitor to transition to the Open Access state as a result of speculation, or from some other cause. This is in addition to the transitions to Open Access state caused by the architectural execution of an operation shown in Figure B1-6.

An implementation must ensure that:

- The local monitor cannot be seen to transition to the Exclusive Access state except as a result of the architectural execution of one of the operations shown in Figure B1-6.
- Any transition of the local monitor to the Open Access state not caused by the architectural execution of an operation shown in Figure B1-6 must not indefinitely delay forward progress of execution.

B1.11.2 Exclusive access instructions and Shareable memory locations

In the context of this section, a shareable memory location is a memory location that has, or is treated as if it has, a Shareability attribute of Inner Shareable or Outer Shareable.

For shareable memory locations, exclusive access instructions rely on:

- A *local monitor* for each PE in the system, which marks any address from which the PE executes a Load-Exclusive. The local monitor operates as described in Exclusive access instructions and Non-shareable memory locations, except that for shareable memory any Store-Exclusive is then subject to checking by the global monitor if it is described in that section as doing at least one of the following:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— Updating memory.
— Returning a status value of 0.

The local monitor can ignore accesses from other PEs in the system.

- A *global monitor* that marks a PA as exclusive access for a particular PE. This marking is used later to determine whether a Store-Exclusive to that address that has not been failed by the local monitor can occur. Any successful write to the marked block by any other observer in the shareability domain of the memory location is guaranteed to clear the marking. For each PE in the system, the global monitor:

  — Can hold at least one marked block.
  — Maintains a state machine for each marked block it can hold.

  Note

  For each PE, the architecture only requires global monitor support for a single marked address. Any situation that might benefit from the use of multiple marked addresses on a single PE is UNPREDICTABLE or CONSTRAINED UNPREDICTABLE, see Load-Exclusive and Store-Exclusive instruction usage restrictions.

Note

The global monitor can either reside within the PE, or exist as a secondary monitor at the memory interfaces. The IMPLEMENTATION DEFINED aspects of the monitors mean that the global monitor and local monitor can be combined into a single unit, provided that the unit performs the global monitor and local monitor functions defined in this manual.

For shareable memory locations, in some implementations and for some memory types, the properties of the global monitor require functionality outside the PE. Some system implementations might not implement this functionality for all locations of memory. In particular, this can apply to:

- Any type of memory in the system implementation that does not support hardware cache coherency.
- Non-cacheable memory, or memory treated as Non-cacheable, in an implementation that does support hardware cache coherency.

In such a system, it is defined by the system:

- Whether the global monitor is implemented.
- If the global monitor is implemented, which address ranges or memory types it monitors.

Note

To support the use of the Load-Exclusive/Store-Exclusive mechanism when address translation is disabled, a system might define at least one location of memory, of at least the size of the translation granule, in the system memory map to support the global monitor for all Arm PEs within a common Inner Shareable domain. However, this is not an architectural requirement. Therefore, architecturally-compliant software that requires mutual exclusion must not rely on using the Load-Exclusive/Store-Exclusive mechanism, and must instead use a software algorithm such as Lamport's Bakery algorithm to achieve mutual exclusion.

Because implementations can choose which memory types are treated as Non-cacheable, the only memory types for which it is architecturally guaranteed that a global Exclusives monitor is implemented are:

- Inner Shareable, Inner Write-Back, Outer Write-Back Normal memory.
- Outer Shareable, Inner Write-Back, Outer Write-Back Normal memory.

The architecture only requires that Conventional memory mapped in this way supports this functionality.

If the global monitor is not implemented for an address range or memory type, then performing a Load-Exclusive or a Store-Exclusive instruction to such a location has one or more of the following effects:

- The instruction generates an External abort.
- The instruction generates an IMPLEMENTATION DEFINED MMU fault. This is reported using the Data Abort Fault status code of ESR_ELx.DFSC = `110101`.

  If the IMPLEMENTATION DEFINED MMU fault is generated for the EL1&0 translation regime then:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - If the fault is generated because of the memory type defined in the first stage of translation, or if the second stage of translation is disabled, then this is a first stage fault and the exception is taken to EL1.
  - Otherwise, the fault is an MPU fault and the exception is taken to EL2.

  The priority of this fault is IMPLEMENTATION DEFINED.
- The instruction is treated as a NOP.
- The Load-Exclusive instruction is treated as if it were accessing a Non-shareable location, but the state of the local monitor becomes UNKNOWN.
- The Store-Exclusive instruction is treated as if it were accessing a Non-shareable location, but the state of the local monitor becomes UNKNOWN. In this case, if the store exclusive instruction is a store exclusive pair of 64-bit quantities, then the two quantities being stored might not be stored atomically.
- The value held in the result register of the Store-Exclusive instruction becomes UNKNOWN.

In addition, for write transactions generated by non-PE observers that do not implement exclusive accesses or other atomic access mechanisms, the effect that writes have on the global and local monitors used by Arm PEs is IMPLEMENTATION DEFINED. The writes might not clear the global monitors of other PEs for:

- Some address ranges.
- Some memory types.

B1.11.2.1 Operation of the global Exclusives monitor

A Load-Exclusive instruction from shareable memory performs a load from memory, and causes the PA of the access to be marked as exclusive access for the requesting PE. This access can also cause the exclusive access mark to be removed from any other PA that has been marked by the requesting PE.

Note

The global monitor supports only a single outstanding exclusive access to shareable memory per PE.

A Load-Exclusive instruction by one PE has no effect on the global monitor state for any other PE.

A Store-Exclusive instruction performs a conditional store to memory:

- The store is guaranteed to succeed only if the PA accessed is marked as exclusive access for the requesting PE and both the local monitor and the global monitor state machines for the requesting PE are in the Exclusive Access state. In this case:
  - A status value of 0 is returned to a register to acknowledge the successful store.
  - The final state of the global monitor state machine for the requesting PE is IMPLEMENTATION DEFINED.
  - If the address accessed is marked for exclusive access in the global monitor state machine for any other PE, then that state machine transitions to Open Access state.
- If no address is marked as exclusive access for the requesting PE, the store does not succeed:
  - A status value of 1 is returned to a register to indicate that the store failed.
  - The global monitor is not affected and remains in Open Access state for the requesting PE.
- If a different PA is marked as exclusive access for the requesting PE, it is IMPLEMENTATION DEFINED whether the store succeeds or not:
  - If the store succeeds a status value of 0 is returned to a register, otherwise a value of 1 is returned.
  - If the global monitor state machine for the PE was in the Exclusive Access state before the Store-Exclusive instruction it is IMPLEMENTATION DEFINED whether that state machine transitions to the Open Access state.

The Store-Exclusive instruction defines the register to which the status value is returned.

In a shared memory system, the global monitor implements a separate state machine for each PE in the system. The state machine for accesses to shareable memory by PE(n) can respond to all the shareable memory accesses visible to it. This means that it responds to:

- Accesses generated by PE(n).

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Accesses generated by the other observers in the shareability domain of the memory location. These accesses are identified as (!n).

In a shared memory system, the global monitor implements a separate state machine for each observer that can generate a Load-Exclusive or a Store-Exclusive instruction in the system.

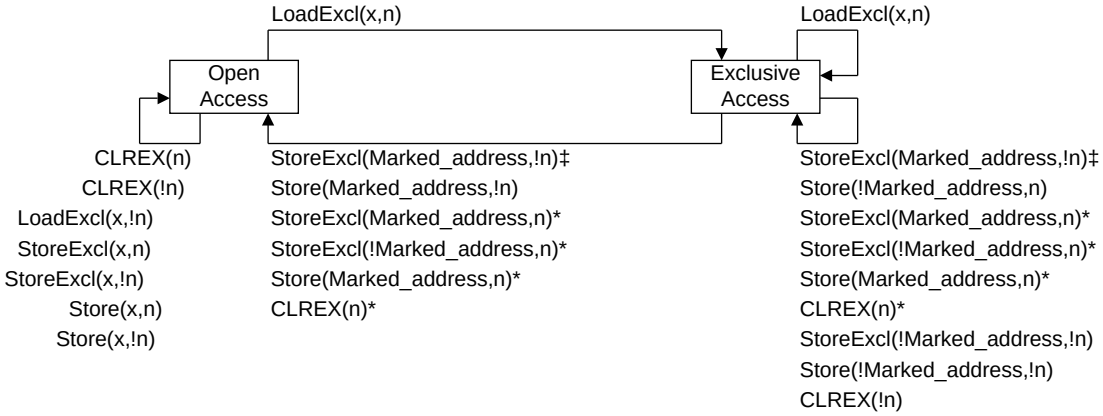
A global monitor:

- In the Exclusive Access state is *set*.
- In the Open Access state is *clear*.

B1.11.2.1.1 Clear global monitor event

Whenever the global monitor state for a PE changes from Exclusive access to Open access, an event is generated and held in the Event register for that PE. This register is used by the Wait for Event mechanism, see Mechanisms for entering a low-power state.

Figure B1-7 shows the state machine for PE(n) in a global monitor.



‡StoreExcl(Marked_address,!n) clears the monitor only if the StoreExcl updates memory
Operations marked * are possible alternative IMPLEMENTATION DEFINED options.
In the diagram: LoadExcl represents any Load-Exclusive instruction
StoreExcl represents any Store-Exclusive instruction
Store represents any other store instruction.

Any LoadExcl operation updates the marked address to the most significant bits of the address x used for the operation.

Figure B1-7 Global monitor state machine diagram for PE(n) in a multiprocessor system

For more information about marking, see Marking and the size of the marked memory block.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

For the global monitor state machine, as shown in Figure B1-7:

- The architecture does not require a load instruction by another PE, that is not a Load-Exclusive instruction, to have any effect on the global monitor.
- Whether a Store-Exclusive instruction successfully updates memory or not depends on whether the address accessed matches the marked shareable memory address for the PE issuing the Store-Exclusive instruction, and whether the local and global monitors are in the exclusive state. For this reason, Figure B1-7 shows only how the operations by (!n) cause state transitions of the state machine for PE(n).
- A Load-Exclusive instruction can only update the marked shareable memory address for the PE issuing the Load-Exclusive instruction.
- When the global monitor is in the Exclusive Access state, it is IMPLEMENTATION DEFINED whether a `CLREX` instruction causes the global monitor to transition from Exclusive Access to Open Access state.
- It is IMPLEMENTATION DEFINED:
  - Whether a modification to a Non-shareable memory location can cause a global monitor to transition from Exclusive Access to Open Access state.
  - Whether a Load-Exclusive instruction to a Non-shareable memory location can cause a global monitor to transition from Open Access to Exclusive Access state.

B1.11.3 Marking and the size of the marked memory block

When a Load-Exclusive instruction is executed, the resulting marked block ignores the least significant bits of the 64-bit memory address.

When a Load-Exclusive instruction is executed, a marked block of size 2^a bytes is created by ignoring the least significant bits of the memory address. A marked address is any address within this marked block. The size of the marked memory block is called the *Exclusives reservation granule*. The Exclusives reservation granule is IMPLEMENTATION DEFINED in the range 4-512 words.

Note

This definition means that the Exclusives reservation granule is:

- 4 words in an implementation where *a* is 4.
- 512 words in an implementation where *a* is 11.

For example, in an implementation where *a* is 4, a successful `LDXRB` of address `0x341B4` defines a marked block using bits[47:4] of the address. This means that the four words of memory from `0x341B0` to `0x341BF` are marked for exclusive access.

In some implementations the CTR identifies the Exclusives reservation granule, see CTR_EL0. Otherwise, software must assume that the maximum Exclusives reservation granule, 512 words, is implemented.

B1.11.4 Context switch support

An exception return clears the local monitor. As a result, performing a `CLREX` instruction as part of a context switch is not required in most situations.

Note

Context switching is not an application level operation. However, this information is included here to complete the description of the exclusive operations.

B1.11.5 Load-Exclusive and Store-Exclusive instruction usage restrictions

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The Load-Exclusive and Store-Exclusive instructions are intended to work together as a pair, for example a `LDXP/STXP` pair or a `LDXR/STXR` pair, executed as part of a loop that contains only a single Load-Exclusive and Store-Exclusive pair. To support different implementations of these functions, software must follow the notes and restrictions given here.

The following notes describe the use of a `LoadExcl/StoreExcl` instruction pair, to indicate the use of any of the Load-Exclusive/Store-Exclusive instruction pairs shown in Table B1-5. In this context, a `LoadExcl/StoreExcl` pair comprises two instructions in the same thread of execution:

- The exclusives support a single outstanding exclusive access for each PE thread that is executed. The architecture makes use of this by not requiring an address or size check as part of the `IsExclusiveLocal()` function. If the target VA of a `StoreExcl` is different from the VA of the preceding `LoadExcl` instruction in the same thread of execution, behavior can be CONSTRAINED UNPREDICTABLE with the following behavior:
  - The `StoreExcl` either passes or fails, the status value returned by the `StoreExcl` is UNKNOWN, and the states of the local and global monitors for that PE are UNKNOWN.

    Note

    This means the `StoreExcl` might pass for some instances of a `LoadExcl/StoreExcl` pair with mismatched addresses, and fail for other instances of a `LoadExcl/StoreExcl` pair with mismatched addresses.

  - The data at the address accessed by the `LoadExcl`, and at the address accessed by the `StoreExcl`, is UNKNOWN.

  This means software can rely on a `LoadExcl/StoreExcl` pair to eventually succeed only if the `LoadExcl` and the `StoreExcl` are executed with the same VA.
- An implementation of the Load-Exclusive and Store-Exclusive instructions can require that, in any thread of execution, the transaction size of a `StoreExcl` instruction is the same as the transaction size of the preceding `LoadExcl` instruction executed in that thread. If the transaction size of a `StoreExcl` instruction is different from the preceding `LoadExcl` instruction in the same thread of execution, behavior can be CONSTRAINED UNPREDICTABLE with the following behavior:
  - The `StoreExcl` either passes or fails, and the status value returned by the `StoreExcl` is UNKNOWN.

    Note

    This means the `StoreExcl` might pass for some instances of a `LoadExcl/StoreExcl` pair with mismatched transaction sizes, and fail for other instances of a `LoadExcl/StoreExcl` pair with mismatched transaction sizes.

  - The block of data of the size of the larger of the transaction sizes used by the `LoadExcl/StoreExcl` pair at the address accessed by the `LoadExcl/StoreExcl` pair, is UNKNOWN.

  This means software can rely on a `LoadExcl/StoreExcl` pair to eventually succeed only if the `LoadExcl` and the `StoreExcl` have the same transaction size.
- An implementation of the `LoadExcl` and `StoreExcl` instructions can require that, in any thread of execution, the `StoreExcl` instruction accesses the same number of registers as the preceding `LoadExcl` instruction executed in that thread. If the `StoreExcl` instruction accesses a different number of registers than the preceding `LoadExcl` instruction in the same thread of execution, behavior is CONSTRAINED UNPREDICTABLE. As a result, software can rely on an `LoadExcl/StoreExcl` pair to eventually succeed only if they access the same number of registers. For more information, see CONSTRAINED UNPREDICTABLE behavior when Load-Exclusive/Store-Exclusive access a different number of registers.
- If FEAT_LSE is not implemented, `LoadExcl/StoreExcl` loops are guaranteed to make forward progress only if, for any `LoadExcl/StoreExcl` loop within a single thread of execution, the software meets all of the following conditions. If FEAT_LSE is implemented, software can maximize the likelihood that `LoadExcl/StoreExcl` loops make forward progress, for any `LoadExcl/StoreExcl` loop within a single thread of execution, by meeting all of the following conditions:

  1 Between the Load-Exclusive and the Store-Exclusive, there are no explicit memory effects, software prefetches, direct or indirect System register writes, address translation instructions, cache or TLB maintenance instructions, exception generating instructions, exception returns, `ISB` barriers, indirect branches, or Branch with Link instructions.

  2 Between the Store-Exclusive returning a failing result and the retry of the corresponding Load-Exclusive:
  - There are no stores or `PRFM` or `RPRFM` instructions to any address within the Exclusives reservation granule accessed by the Store-Exclusive. This includes VA aliases to those addresses.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— There are no loads or prefetches to any address within the Exclusives reservation granule accessed by the Store-Exclusive that use a different VA alias to that address.

— There are no other Store-Exclusive instructions to any other address.

— There are no direct or indirect System register writes, address translation instructions, cache or TLB maintenance instructions, exception generating instructions, exception returns, indirect branches, or Branch with Link instructions.

— All loads and stores are to a block of contiguous virtual memory of not more than 512 bytes in size.

The Exclusives monitor can be cleared at any time without an application-related cause, provided that such clearing is not systematically repeated so as to prevent the forward progress in finite time of at least one of the threads that is accessing the Exclusives monitor. However, it is permissible for the `LoadExcl`/`StoreExcl` loop not to make forward progress if a different thread is repeatedly doing any of the following in a tight loop:

— Performing stores to a PA covered by the Exclusives monitor.

— Prefetching with intent to write to a PA covered by the Exclusives monitor.

— Executing data cache clean, data cache invalidate, or data cache clean and invalidate instructions to a PA covered by the Exclusives monitor.

— Executing instruction cache invalidate all instructions.

— Executing instruction cache invalidate by VA instructions to a PA covered by the Exclusives monitor.

— Executing TLB maintenance to a PA covered by the Exclusives monitor.

- Implementations can benefit from keeping the `LoadExcl` and `StoreExcl` operations close together in a single thread of execution. This minimizes the likelihood of the Exclusives monitor state being cleared between the `LoadExcl` instruction and the `StoreExcl` instruction. Therefore, for best performance, Arm strongly recommends a limit of 128 bytes between `LoadExcl` and `StoreExcl` instructions in a single thread of execution.
- The architecture sets an upper limit of 2048 bytes on the Exclusives reservation granule that can be marked as exclusive. For performance reasons, Arm recommends that objects that are accessed by exclusive accesses are separated by the size of the Exclusives reservation granule. This is a performance guideline rather than a functional requirement.
- After taking a Data Abort exception, the state of the Exclusives monitors is UNKNOWN.
- For the memory location accessed by a `LoadExcl`/`StoreExcl` pair, if the memory attributes for a `StoreExcl` instruction are different from the memory attributes for the preceding `LoadExcl` instruction in the same thread of execution, behavior is CONSTRAINED UNPREDICTABLE. Where this occurs because the translation of the accessed address changes between the `LoadExcl` instruction and the `StoreExcl` instruction, the CONSTRAINED UNPREDICTABLE behavior is as follows:

  — The `StoreExcl` either passes or fails, and the status value returned by the `StoreExcl` is UNKNOWN.

  Note

  This means the `StoreExcl` might pass for some instances of a `LoadExcl`/`StoreExcl` pair with changed memory attributes, and fail for other instances of a `LoadExcl`/`StoreExcl` pair with changed memory attributes.

  — The data at the address accessed by the `StoreExcl` is UNKNOWN.

  Note

  Another bullet point in this list covers the case where the memory attributes of a `LoadExcl`/`StoreExcl` pair differ as a result of using different VAs with different attributes that point to the same PA.

- The effect of a data or unified cache invalidate, clean, or clean and invalidate instruction on a local or global Exclusives monitor that is in the Exclusive Access state is CONSTRAINED UNPREDICTABLE, and the instruction might clear the monitor, or it might leave it in the Exclusive Access state. For address-based maintenance instructions, this also applies to the monitors of other PEs in the same shareability domain as the PE executing the cache maintenance instruction, as determined by the shareability domain of the address being maintained.

  Note

  Arm strongly recommends that implementations ensure that the use of such maintenance instructions by a PE in one Security state cannot cause a denial of service on a PE in another Security state.

- If the mapping of the VA to PA is changed between the `LoadExcl` instruction and the `StoreExcl` instruction, and the change is performed using a break-before-make sequence as described in Using break-before-make when

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

updating translation table entries, if the `StoreExcl` is performed after another write to the same PA as the `StoreExcl`, and that other write was performed after the old translation was properly invalidated and that invalidation was properly synchronized, then the `StoreExcl` will not pass its monitor check.

Note

Arm expects that, in many implementations, either:

— The TLB invalidation will clear either the local or global monitor.

— The PA will be checked between the `LoadExcl` and `StoreExcl`.

B1.11.5.1 CONSTRAINED UNPREDICTABLE behavior when Load-Exclusive/Store-Exclusive access a different number of registers

As stated in this section, an implementation can require that the instructions of a Load-Exclusive/Store-Exclusive pair access the same number of registers. In such an implementation, this means behavior is CONSTRAINED UNPREDICTABLE if, in a single thread of execution, either:

- An `LDXP` instruction of two 32-bit quantities is followed by an `STXR` instruction of one 64-bit quantity at the same address.
- An `LDXR` instruction of one 64-bit quantity is followed by an `STXP` instruction of two 32-bit quantities at the same address.

In these cases, the CONSTRAINED UNPREDICTABLE behavior must be one of:

- The `STXP` or `STXR` instruction generates an external Data Abort.
- The `STXP` or `STXR` instruction generates an IMPLEMENTATION DEFINED MMU fault reported using the Data Abort Fault status code of ESR_ELx.DFSC = `0b110101`.
- The `STXP` or `STXR` instruction always fails, returning a status of 1.
- The `STXP` or `STXR` instruction always passes, returning a status of 0.
- This `STXP` or `STXR` instruction has the same pass or fail behavior that it would have had if the instruction had used the same size and number of registers as the preceding `LDXR` or `LDXP` instruction.

B1.11.6 Use of WFE and SEV instructions by spin-locks

Armv8 provides Wait For Event, Send Event, and Send Event Local instructions, `WFE`, `SEV`, and `SEVL`, that can assist with reducing power consumption and bus contention caused by PEs repeatedly attempting to obtain a spin-lock. These instructions can be used at the application level, but a complete understanding of what they do depends on a system level understanding of exceptions. They are described in Wait for Event. However, in Armv8, when the global monitor for a PE changes from Exclusive Access state to Open Access state, an event is generated.

Note

This is equivalent to issuing an `SEVL` instruction on the PE for which the monitor state has changed. It removes the need for spinlock code to include an `SEV` instruction after clearing a spinlock.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Part C
The AArch64 Instruction Set

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter C1
The A64 Instruction Set

This chapter describes the A64 instruction set. It contains the following sections:

- About the A64 instruction set.
- Structure of the A64 assembler language.
- Address generation.
- Instruction aliases.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C1.1 About the A64 instruction set

The A64 instruction set is the instruction set supported in the AArch64 Execution state. All A64 instructions have a width of 32 bits.

The following chapters contain alphabetical lists of A64 instructions:

- A64 Base Instruction Descriptions lists all of:
  - Branch instructions, exception generating instructions, system instructions.
  - Load, store, and data-processing instructions associated with the general-purpose registers.
- A64 Advanced SIMD and Floating-point Instruction Descriptions lists the load, store, and data-processing instructions associated with Advanced SIMD and floating-point support.

When an instruction supports more than one syntax, each syntax is an *instruction variant*. Instruction variants can occur because of differences in:

- The size or format of the operands.
- The register file used for the operands.
- The addressing mode used for load/load/store memory operands.

Instruction variants might also arise as the result of other factors.

Instruction variants are described in the instruction description for the individual instructions.

A64 instruction set encoding describes the A64 encoding structure. A64 instructions have a regular bit encoding structure:

- 5-bit register operand fields at fixed positions within the instruction. For general-purpose register operands, the values 0-30 select one of 31 registers. The value 31 is used as a special case that can:
  - Indicate use of the current stack pointer, when identifying a load/store base register or in a limited set of data-processing instructions. See The stack pointer registers.
  - Indicate the value zero when used as a source register operand.
  - Indicate discarding the result when used as a destination register operand.
- For Advanced SIMD and floating-point register access, a 5-bit register operand field selects one of 32 registers of the appropriate type.
- Certain Advanced SIMD instructions have a register operand field smaller than 5 bits when specifying an indexed vector element or multi-vector operand, which restricts the number of vector registers that can be accessed by those operands.
- Immediate bits that provide constant data-processing values or address offsets are placed in contiguous bitfields. Some computed values in instruction variants use one or more immediate bitfields together with the secondary encoding bitfields.

All encodings that are not fully defined are described as unallocated. An attempt to execute an unallocated instruction is UNDEFINED, unless the behavior is otherwise defined in this Manual.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C1.2 Structure of the A64 assembler language

The following sections describe the A64 assembler syntax:

- General requirements.
- Common syntax terms.
- Instruction Mnemonics.
- Condition code.
- Register names.

C1.2.1 General requirements

The letter `W` denotes a general-purpose register holding a 32-bit word, and `X` denotes a general-purpose register holding a 64-bit doubleword.

An A64 assembler recognizes both uppercase and lowercase variants of the instruction mnemonics and register names, but not mixed case variants. An A64 disassembler can output either uppercase or lowercase mnemonics and register names. Program and data labels are case-sensitive.

The A64 assembly language does not require the # character to introduce constant immediate operands, but an assembler must allow immediate values introduced with or without the # character.

In Example C1-1, the sequence // is used as a comment leader and A64 assemblers are encouraged to accept this syntax.

C1.2.2 Common syntax terms

The following syntax terms are used frequently throughout the A64 instruction set description.

`UPPER` Text in upper-case letters is fixed. Text in lower-case letters is variable. This means that register name `Xn` indicates that the `X` is required, followed by a variable register number, for example `X29`.

`< >` Any text enclosed by angle braces, < >, is a value that the user supplies. Subsequent text might supply additional information.

`{ }` Any item enclosed by curly brackets, { }, is optional. A description of the item and how its presence or absence affects the instruction is normally supplied by subsequent text. In some cases curly braces are actual symbols in the syntax, for example when they surround a register list. These cases are called out in the surrounding text.

`[ ]` Any items enclosed by square brackets, [ ], constitute a list of alternative characters. A single one of the characters can be used in that position and the subsequent text describes the meaning of the alternatives. In some case the square brackets are part of the syntax itself, such as addressing modes or vector elements. These cases are called out in the surrounding text.

`a|b` Alternative words are separated by a vertical bar, |, and can be surrounded by parentheses to delimit them. For example, `U(ADD|SUB)W` represents `UADDW` or `USUBW`.

`±` This indicates an optional + or - sign. If neither is used then + is assumed.

`uimmn` An *n*-bit unsigned, positive, immediate value.

`simmn` An *n*-bit two's complement, signed immediate value, where *n* includes the sign bit.

`SP` See Register names.

`Wn` See Register names.

`WSP` See Register names.

`WZR` See Register names.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`Xn`	See Register names.
`XZR`	See Register names

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C1.2.3 Instruction Mnemonics

The A64 assembly language overloads instruction mnemonics and distinguishes between the different forms of an instruction based on the operand types. For example, the following `ADD` instructions all have different opcodes. However, the programmer must remember only one mnemonic, as the assembler automatically chooses the correct opcode based on the operands. The disassembler follows the same procedure in reverse.

Example C1-1 ADD instructions with different opcodes

```
ADD W0, W1, W2          // add 32-bit register
ADD X0, X1, X2          // add 64-bit register
ADD X0, X1, W2, SXTW    // add 64-bit extended register
ADD X0, X1, #42         // add 64-bit immediate
```

C1.2.4 Condition code

The A64 ISA has some instructions that set Condition flags or test Condition codes or both. For information about instructions that set the Condition flags or use the condition mnemonics, see Condition flags and related instructions.

Table C1-1 shows the available Condition codes.

Table C1-1 Condition codes

cond	Mnemonic	Meaning (integer)	Meaning (floating-point)[a]	Condition flags
0000	EQ	Equal	Equal	Z == 1
0001	NE	Not equal	Not equal or unordered	Z == 0
0010	CS or HS	Carry set	Greater than, equal, or unordered	C == 1
0011	CC or LO	Carry clear	Less than	C == 0
0100	MI	Minus, negative	Less than	N == 1
0101	PL	Plus, positive or zero	Greater than, equal, or unordered	N == 0
0110	VS	Overflow	Unordered	V == 1
0111	VC	No overflow	Ordered	V == 0
1000	HI	Unsigned higher	Greater than, or unordered	C ==1 && Z == 0
1001	LS	Unsigned lower or same	Less than or equal	!(C ==1 && Z ==0)
1010	GE	Signed greater than or equal	Greater than or equal	N == V
1011	LT	Signed less than	Less than, or unordered	N != V
1100	GT	Signed greater than	Greater than	Z == 0 && N == V
1101	LE	Signed less than or equal	Less than, equal, or unordered	!(Z == 0 && N == V)
1110	AL	Always	Always	Any
1111	NV[b]	Always	Always	Any

a. Unordered means at least one NaN operand.

b. The Condition code `NV` exists only to provide a valid disassembly of the `0b1111` encoding, otherwise its behavior is identical to `AL`.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C1.2.5 Register names

See:

- General-purpose register file and zero register and stack pointer.
- Advanced SIMD and floating-point register file.
- Advanced SIMD and floating-point scalar register names.
- SIMD vector register names.
- SIMD vector element names.

C1.2.5.1 General-purpose register file and zero register and stack pointer

The 31 general-purpose registers in the general-purpose register file are named R0-R30 and encoded in the instruction register fields with values 0-30. In a general-purpose register field the value 31 represents either the current stack pointer or the zero register, depending on the instruction and the operand position.

When the registers are used in a specific instruction variant, they must be qualified to indicate the operand data size, 32 bits or 64 bits, and the data size of the instruction.

When the data size is 32 bits, the lower 32 bits of the register are used and the upper 32 bits are ignored on a read and cleared to zero on a write.

Table C1-2 shows the qualified names for registers, where *n* is a register number 0-30.

Table C1-2 Naming of general-purpose registers, the zero register, and the stack pointer

Name	Size	Encoding	Description
Wn	32 bits	0-30	General-purpose register 0-30
Xn	64 bits	0-30	General-purpose register 0-30
WZR	32 bits	31	Zero register
XZR	64 bits	31	Zero register
WSP	32 bits	31	Current stack pointer
SP	64 bits	31	Current stack pointer

This list gives more information about the instruction arguments shown in Table C1-2:

- The names Xn and Wn both refer to the same general-purpose register, Rn.
- There is no register named W31 or X31.
- The name SP represents the stack pointer for 64-bit operands where an encoding of the value 31 in the corresponding register field is interpreted as a read or write of the current stack pointer. When instructions do not interpret this operand encoding as the stack pointer, use of the name SP is an error.
- The name WSP represents the current stack pointer in a 32-bit context.
- The name XZR represents the zero register for 64-bit operands where an encoding of the value 31 in the corresponding register field is interpreted as returning zero when read or discarding the result when written. When instructions do not interpret this operand encoding as the zero register, use of the name XZR is an error.
- The name WZR represents the zero register in a 32-bit context.
- The architecture does not define a specific name for general-purpose register R30 to reflect its role as the link register on procedure calls. However, an A64 assembler must always use W30 and X30 for this purpose, and additional software names might be defined as part of the Procedure Call Standard, see *Procedure Call Standard for the Arm 64-bit Architecture*.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C1.2.5.2 Advanced SIMD and floating-point register file

The 32 registers in the Advanced SIMD and floating-point register file, V0-V31, hold floating-point operands for the scalar floating-point instructions, and both scalar and vector operands for the Advanced SIMD instructions. When they are used in a specific instruction form, the names must be further qualified to indicate the data shape, that is the data element size and the number of elements or lanes within the register. A similar requirement is placed on the general-purpose registers. See General-purpose register file and zero register and stack pointer.

Note

The data type is described by the instruction mnemonics that operate on the data. The data type is not described by the register name. The data type is the interpretation of bits within each register or vector element, whether these are integers, floating-point values, polynomials, or cryptographic hashes.

C1.2.5.3 Advanced SIMD and floating-point scalar register names

Advanced SIMD and floating-point instructions that operate on scalar data only access the lower bits of an Advanced SIMD and floating-point register. The unused high bits are ignored on a read and cleared to 0 on a write.

Table C1-3 shows the qualified names for accessing scalar Advanced SIMD and floating-point registers. The letter *n* denotes a register number between 0 and 31.

Table C1-3 Advanced SIMD and floating-point scalar register names

Size	Name
8 bits	`Bn`
16 bits	`Hn`
32 bits	`Sn`
64 bits	`Dn`
128 bits	`Qn`

C1.2.5.4 SIMD vector register names

If a register holds multiple data elements on which arithmetic is performed in a parallel, SIMD manner, then a qualifier describes the vector shape. The vector shape is the element size and the number of elements or lanes. If the element size in bits multiplied by the number of lanes does not equal 128, then the upper 64 bits of the register are ignored on a read and cleared to zero on a write.

Table C1-4 shows the SIMD vector register names. The letter *n* denotes a register number between 0 and 31.

Table C1-4 SIMD vector register names

Shape	Name
8 bits × 8 lanes	`Vn.8B`
8 bits × 16 lanes	`Vn.16B`
16 bits × 4 lanes	`Vn.4H`
16 bits × 8 lanes	`Vn.8H`
32 bits × 2 lanes	`Vn.2S`
32 bits × 4 lanes	`Vn.4S`

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Shape	Name
64 bits × 1 lane	`Vn.1D`
64 bits × 2 lanes	`Vn.2D`

C1.2.5.5 SIMD vector element names

Appending a constant, zero-based element index to the register name inside square brackets indicates that a single element from a SIMD and floating-point register is used as a scalar operand. The number of lanes is not represented, as it is not encoded in the instruction and can only be inferred from the index value.

Table C1-5 shows the vector register names and the element index. The letter *i* denotes the element index.

Table C1-5 Vector register names with element index

Size	Name
8 bits	`Vn.B[i]`
16 bits	`Vn.H[i]`
32 bits	`Vn.S[i]`
64 bits	`Vn.D[i]`

An assembler must accept a fully qualified SIMD register name if the number of lanes is greater than the index value. See SIMD vector register names. For example, an assembler must accept all of the following forms as the name for the 32-bit element in bits [63:32] of the SIMD and floating-point register V9:

```
V9.S[1]     //standard disassembly
V9.2S[1]    //optional number of lanes
V9.4S[1]    //optional number of lanes
```

Note

The SIMD and floating-point register element name `Vn.S[0]` is not equivalent to the scalar SIMD and floating-point register name `Sn`. Although they represent the same bits in the register, they select different instruction encoding forms, either the vector element or the scalar form.

C1.2.5.5.1 SIMD vector register list

Where an instruction operates on multiple SIMD&FP, for example vector load/store structure and table lookup operations, the registers are specified as a list enclosed by curly braces. This list consists of either a sequence of registers separated by commas, or a register range separated by a hyphen. The registers must be numbered in increasing order, modulo 32, in increments of one. The hyphenated form is preferred for disassembly if there are more than two registers in the list and the register numbers are increasing by 1. The following examples are equivalent representations of a set of four registers `V4` to `V7`, each holding four lanes of 32-bit elements:

```
{ V4.4S - V7.4S }              //standard disassembly
{ V4.4S, V5.4S, V6.4S, V7.4S } //alternative representation
```

C1.2.5.5.2 SIMD vector element list

Registers in a list can also have a vector element form. For example, the `LD4` instruction can load one element into each of four registers, and in this case the index is appended to the list as follows:

```
{ V4.S - V7.S }[3]                //standard disassembly
{ V4.4S, V5.4S, V6.4S, V7.4S }[3] //alternative with optional number of lanes
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C1.3 Address generation

The A64 instruction set supports 64-bit virtual addresses (VAs). The valid VA range is determined by the following factors:

- The size of the implemented virtual address space.
- *Memory Management Unit* (MMU) configuration settings.
- *Memory Protection Unit* (MPU) configuration settings.

Limits on the VA size mean that the most significant bits of the virtual address do not hold valid address bits. These unused bits can hold:

- A tag, see Address tagging.
- If FEAT_PAuth is implemented, a Pointer authentication code (PAC), see Pointer authentication.

For more information on memory management and address translation, see The AArch64 Virtual Memory System Architectureand The AArch64 Protected Memory System Architecture.

C1.3.1 Register indexed addressing

The A64 instruction set allows a 64-bit index register to be added to the 64-bit base register, with optional scaling of the index by the access size. Additionally it allows for sign-extension or zero-extension of a 32-bit value within an index register, followed by optional scaling.

C1.3.2 PC-relative addressing

The A64 instruction set has support for position-independent code and data addressing:

- PC-relative literal loads have an offset range of ± 1MB.
- Process state flag and compare based conditional branches have a range of ± 1MB. Test bit conditional branches have a restricted range of ± 32KB.
- Unconditional branches, including branch and link, have a range of ± 128MB.

PC-relative load/store operations, and address generation with a range of ± 4GB can be performed using two instructions.

C1.3.3 Load/store addressing modes

Load/store addressing modes in the A64 instruction set require a 64-bit base address from a general-purpose register X0-X30 or the current stack pointer, `SP`, with an optional immediate or register offset. Table C1-6 shows the assembler syntax for the complete set of load/store addressing modes.

Table C1-6 A64 Load/store addressing modes

Addressing Mode	Offset		
	Immediate	Register	Extended Register
Base register only (no offset)	`[base{, #0}]`	-	-
Base plus offset	`[base{, #imm}]`	`[base, Xm{, LSL #imm}]`	`[base, Wm, (S|U)XT(X|W) {#imm}]`
Pre-indexed	`[base, #imm]!`	-	-

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Addressing Mode	Offset		
	Immediate	Register	Extended Register
Post-indexed	`[base], #imm`	`[base], Xm`[a]	-
Literal (PC-relative)	`label`	-	-

a. The post-indexed by register offset mode can be used with the SIMD load/store structure instructions described in Load/store Advanced SIMD. Otherwise the post-indexed by register offset mode is not available.

Some types of load/store instruction support only a subset of the load/store addressing modes listed in Table C1-6. Details of the supported modes are as follows:

- Base plus offset addressing means that the address is the value in the 64-bit base register plus an offset.
- Pre-indexed addressing means that the address is the sum of the value in the 64-bit base register and an offset, and the address is then written back to the base register.
- Post-indexed addressing means that the address is the value in the 64-bit base register, and the sum of the address and the offset is then written back to the base register.
- Literal addressing means that the address is the value of the 64-bit program counter for this instruction plus a 19-bit signed word offset. This means that it is a 4 byte aligned address within ±1MB of the address of this instruction with no offset. Literal addressing can be used only for loads of at least 32 bits and for prefetch instructions. The PC cannot be referenced using any other addressing modes. The syntax for labels is specific to individual toolchains.
- An immediate offset can be unsigned or signed, and scaled or unscaled, depending on the type of load/store instruction. When the immediate offset is scaled it is encoded as a multiple of the transfer size, although the assembly language always uses a byte offset, and the assembler or disassembler performs the necessary conversion. The usable byte offsets therefore depend on the type of load/store instruction and the transfer size. Table C1-7 shows the offset and the type of load/store instruction.

Table C1-7 Immediate offsets and the type of load/store instruction

Offset bits	Sign	Scaling	Write-Back	Load/store type
0	-	-	-	Exclusive/acquire/release
7	Signed	Scaled	Optional	Register pair
9	Signed	Unscaled	Optional	Single register
12	Unsigned	Scaled	No	Single register

- A register offset means that the offset is the 64 bits from a general-purpose register, Xm, optionally scaled by the transfer size, in bytes, if `LSL #imm` is present and where `imm` must be equal to $\log_2$(transfer_size). The `SXTX` extend/shift option is functionally equivalent to `LSL`, but the `LSL` option is preferred in source code.
- An extended register offset means that offset is the bottom 32 bits from a general-purpose register Wm, sign-extended or zero-extended to 64 bits, and then scaled by the transfer size if so indicated by `#imm`, where imm must be equal to $\log_2$(transfer_size). An assembler must accept Wm or Xm as an extended register offset, but Wm is preferred for disassembly.
- Generating an address lower than the value in the base register requires a negative signed immediate offset or a register offset holding a negative value.
- When stack alignment checking is enabled by system software and the base register is the SP, the current stack pointer must be initially quadword aligned, that is aligned to 16 bytes. Misalignment generates a Stack Alignment fault. The offset does not have to be a multiple of 16 bytes unless the specific load/store instruction requires this. SP cannot be used as a register offset.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C1.3.3.1 Address calculation

General-purpose arithmetic instructions can calculate the result of most addressing modes and write the address to a general-purpose register or, in most cases, to the current stack pointer.

Table C1-8 shows the arithmetic instructions that can compute addressing modes.

Table C1-8 Arithmetic instructions to compute addressing modes

Addressing Form	Offset		
	Immediate	Register	Extended Register
Base register (no offset)	`MOV Xd|SP, base`	-	-
Base plus offset	`ADD Xd|SP, base, #imm` or `SUB Xd|SP, base, #imm`	`ADD <Xd|SP>, base, Xm{,LSL#imm}`	`ADD <Xd|SP>, base, Wm,(S|U)XT(W|H|B|X) {#imm}`
Pre-indexed	-	-	-
Post-indexed	-	-	-
Literal (PC-relative)	`ADR Xd, label`	-	-

Note

- For the 64-bit base plus register offset form, the `UXTX` mnemonic is an alias for the `LSL` shift option, but `LSL` is preferred for disassembly. Similarly the `SXTX` extend/shift option is functionally equivalent to the `LSL` option, but the `LSL` option is preferred in source code.
- To calculate a base plus immediate offset the `ADD` instructions defined in Arithmetic (immediate) accept an unsigned 12-bit immediate offset, with an optional left shift by 12. This means that a single `ADD` instruction cannot support the full range of byte offsets available to a single register load/store with a scaled 12-bit immediate offset. For example, a quadword `LDR` effectively has a 16-bit byte offset. To calculate an address with a byte offset that requires more than 12 bits it is necessary to use two `ADD` instructions. The following example shows this:

```
ADD Xd, base, #(imm & 0xFFF)
ADD Xd, Xd, #(imm>>12), LSL #12
```

- To calculate a base plus extended register offset, the `ADD` instructions defined in Arithmetic (extended register) provide a superset of the addressing mode that also supports sign-extension or zero-extension of a byte or halfword value with any shift amount between 0 and 4, for example:

```
ADD  Xd, base, Wm, SXTW #3   // Xd = base + (SignExtend(Wm) LSL 3)
ADD  Xd, base, Wm, UXTH #4   // Xd = base + (ZeroExtend(Wm<15:0>) LSL 4)
```

- If the same extended register offset is used by more than one load/store instruction, then, depending on the implementation, it might be more efficient to calculate the extended and scaled intermediate result just once, and then reuse it as a simple register offset. The extend and scale calculation can be performed using the `SBFIZ` and `UBFIZ` bitfield instructions defined in Bitfield move, for example:

```
SBFIZ Xd, Xm, #3, #32   //Xd = "Wm, SXTW #3"
UBFIZ Xd, Xm, #4, #16   //Xd = "Wm, UXTH #4"
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C1.4 Instruction aliases

Some instructions have an associated *architecture alias* that is used for disassembly of the encoding when the associated conditions are met. Architecture alias instructions are included in the alphabetic lists of instruction types and clearly presented as an alias form in descriptions for the individual instructions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter C2
About the A64 Instruction Descriptions

This chapter describes the *instruction descriptions* contained in A64 Base Instruction Descriptions, and A64 Advanced SIMD and Floating-point Instruction Descriptions.

It contains the following sections:

- Understanding the A64 instruction descriptions.
- General information about the A64 instruction descriptions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C2.1 Understanding the A64 instruction descriptions

Each instruction description in A64 Base Instruction Descriptions and A64 Advanced SIMD and Floating-point Instruction Descriptions has the following content:

1. A title.
2. An introduction to the instruction.
3. The instruction encoding or encodings.
4. Any alias conditions.
5. A list of the assembler symbols for the instruction.
6. Pseudocode describing how the instruction operates.
7. Notes, if applicable.

The following sections describe each of these.

C2.1.1 The title

The title of an instruction description includes the base mnemonic for the instruction.

If different forms of an instruction use the same base mnemonic, each form has its own description. In this case, the title is the mnemonic followed by a short description of the instruction form in parentheses. This is most often used when an operand is an immediate value in one instruction form, but is a register in another form.

For example, in A64 Base Instruction Descriptions there are the following titles for different forms of the `ADD` instruction:

- ADD (extended register).
- ADD (immediate).
- ADD (shifted register).

C2.1.2 An introduction to the instruction

This briefly describes the function of the instruction. The introduction is not a complete description of the instruction, and it is not definitive. If there is any conflict between it and the more detailed information that follows it, the more detailed information takes priority.

C2.1.3 The instruction encoding or encodings

This shows the instruction encoding diagram, or if the instruction has more than one encoding, shows all of the encoding diagrams. Each diagram has a subheading.

For example, for load and store instructions, the subheadings might be:

- Post-index.
- Pre-index.
- Unsigned offset.

Each diagram numbers the bits from 31 to 0. The diagram for an instruction at address *A* shows, from left to right, the bytes at addresses *A*+3, *A*+2, *A*+1, and *A*.

There might be variants of an encoding, if the *assembler syntax prototype* differs depending on the value in one or more of the encoding fields. In this case, each variant has a subheading that describes the variant and shows the distinguishing field value or values in parentheses. For example, in A64 Base Instruction Descriptions there are the following subheadings for variants of the `ADC` instruction encoding:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- 32-bit variant (sf = 0).
- 64-bit variant (sf = 1).

The assembler syntax prototype for an encoding or variant of an encoding shows how to form a complete assembler source code instruction that assembles to the encoding. Unless otherwise stated, the prototype is also the preferred syntax for a disassembler to disassemble the encoding to. Disassemblers are permitted to omit optional symbols that represent the default value of a field or set of fields, to produce more readable disassembled code, provided that the output re-assembles to the same encoding.

Each encoding diagram, and its associated assembler syntax prototypes, is followed by encoding-specific pseudocode that translates the fields of that encoding into inputs for the encoding-independent pseudocode that describes the operation of the instruction. See Pseudocode describing how the instruction operates.

C2.1.4 Any alias conditions, if applicable

This is an optional part of an instruction description. If included, it describes the set of conditions for which an alternative mnemonic and its associated assembler syntax prototypes are preferred for disassembly by a disassembler. It includes a link to the alias instruction description that defines the alternative syntax. The alias syntax and the original syntax can be used interchangeably in the assembler source code.

Arm recommends that if a disassembler outputs the alias syntax, it consistently outputs the alias syntax.

C2.1.5 A list of the assembler symbols for the instruction

The *Assembler symbols* subsection of the instruction description contains a list of the symbols that the assembler syntax prototype or prototypes use, if any.

In assembler syntax prototypes, the following conventions are used:

`< >` Angle brackets. Any symbol enclosed by these is a name or a value that the user supplies. For each symbol, there is a description of what the symbol represents. The description usually also specifies which encoding field or fields encodes the symbol.

`{ }` Brace brackets. Any symbols enclosed by these are optional. For each optional symbol, there is a description of what the symbol represents and how its presence or absence is encoded.

In some assembler syntax prototypes, some brace brackets are mandatory, for example if they surround a register list. When the use of brace brackets is mandatory, they are separated from other syntax items by one or more spaces.

`#` This usually precedes a numeric constant. All uses of # are optional in A64 assembler source code. Arm recommends that disassemblers output the # where the assembler syntax prototype includes it.

`+/-` This indicates an optional + or – sign. If neither is coded, + is assumed.

Single spaces are used for clarity, to separate syntax items. Where a space is mandatory, the assembler syntax prototype shows two or more consecutive spaces.

Any characters not shown in this conventions list must be coded exactly as shown in the assembler syntax prototype. Apart from brace brackets, the characters shown are used as part of a meta-language to define the architectural assembler syntax for an instruction encoding or alias, but have no architecturally defined significance in the input to an assembler or in the output from a disassembler.

The following symbol conventions are used:

`<Xn>` The 64-bit name of a general-purpose register (X0-X30) or the zero register (XZR).

`<Wn>` The 32-bit name of a general-purpose register (W0-W30) or the zero register (WZR).

`<Xn|SP>` The 64-bit name of a general-purpose register (X0-X30) or the current stack pointer (SP).

`<Wn|WSP>` The 32-bit name of a general-purpose register (W0-W30) or the current stack pointer (WSP).

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`<Bn>, <Hn>, <Sn>, <Dn>, <Qn>`

The 8, 16, 32, 64, or 128-bit name of a SIMD and floating-point register in a scalar context, as described in Register names.

`<Vn>` The name of a SIMD and floating-point register in a vector context, as described in Register names.

If the description of a symbol specifies that the symbol is a register, the description might also specify that the range of permitted registers is extended or restricted. It also specifies any differences from the default rules for such fields.

Note

Register names provides the A64 register names.

C2.1.6 Pseudocode describing how the instruction operates

The *Operation* subsection of the instruction description contains this pseudocode.

It is encoding-independent pseudocode that provides a precise description of what the instruction does.

Note

For a description of Arm pseudocode, see Arm Pseudocode Definition. This appendix also describes the execution model for an instruction.

C2.1.7 Notes, if applicable

If applicable, other notes about the instruction appear under additional subheadings.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C2.2 General information about the A64 instruction descriptions

This section provides general information about the A64 instruction descriptions. Some of this information also applies to System register descriptions, for example the terms defined in Fixed values in AArch64 instruction and System register descriptions apply to the AArch64 descriptions throughout this manual. The following subsections provide this information:

- Execution of instructions in Debug state.
- Fixed values in AArch64 instruction and System register descriptions.
- Modified immediate constants in A64 floating-point instructions.

C2.2.1 Execution of instructions in Debug state

In general, except for the instructions described in Debug state, the A64 instruction descriptions do not indicate any differences in the behavior of the instruction if it is executed in Debug state. For this information, see Executing instructions in Debug state.

Note

For many instructions, execution is unchanged in Debug state. Executing instructions in Debug state identifies these instructions,

C2.2.2 Fixed values in AArch64 instruction and System register descriptions

This section summarizes the terms used to describe fixed values in AArch64 register and instruction descriptions. The Glossary gives full descriptions of these terms, and each entry in this section includes a link to the corresponding Glossary entry.

Note

In register descriptions, the meaning of some bits depends on the PE state. This affects the definitions of RES0 and RES1, as shown in the Glossary.

The following terms are used to describe bits or fields with fixed values:

RAZ Read-As-Zero. See Read-As-Zero (RAZ).

In diagrams, a RAZ bit can be shown as 0.

(0), RES0 Reserved, Should-Be-Zero (SBZ) or RES0.

In instruction encoding diagrams, and sometimes in other descriptions, `(0)` indicates an SBZ bit. If the bit is set to 1, behavior is CONSTRAINED UNPREDICTABLE, and must be one of the following:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if the value of the bit was 0.
- Any destination registers of the instruction become UNKNOWN.

This notation can be expanded for fields, so a three-bit field can be shown as either `(0)(0)(0)` or as `(000)`.

In register diagrams, but not in the A64 encoding and instruction descriptions, bits or fields can be shown as RES0. See the Glossary definition of RES0 for more information.

Note

Some of the System instruction descriptions in this chapter are based on the *field description* of the input value for the instruction. These are register descriptions and therefore can include RES0 fields,

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The (0) and RES0 descriptions can be applied to bits or bit fields that are read-only, or are write-only. The Glossary definitions cover these cases.

RAO Read-As-One. See Read-As-One (RAO).

In diagrams, a RAO bit can be shown as 1.

(1), RES1 Reserved, Should-Be-One (SBO) or RES1.

In instruction encoding diagrams, and sometimes in other descriptions, (1) indicates an SBO bit. If the bit is set to 0, behavior is CONSTRAINED UNPREDICTABLE, and must be one of the following:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- The instruction executes as if the value of the bit was 1.
- Any destination registers of the instruction become UNKNOWN.

This notation can be expanded for fields, so a three-bit field can be shown as either (1)(1)(1) or as (111).

In register diagrams, but not in the A64 encoding and instruction descriptions, bits or fields can be shown as RES1. See the Glossary definition of *RES1* for more information.

Note

Some of the System instruction descriptions in this chapter are based on the *field description* of the input value for the instruction. These are register descriptions and therefore can include RES1 fields,

The (1) and RES1 descriptions can be applied to bits or bit fields that are read-only, or are write-only. The Glossary definitions cover these cases.

C2.2.3 Modified immediate constants in A64 floating-point instructions

Table C2-1 shows the immediate constants available in FMOV (scalar, immediate) and FMOV (vector, immediate) floating-point instructions.

Table C2-1 A64 Floating-point modified immediate constants

Data type	immediate	Constant[a]
F16	abcdefgh	aBbbcdef gh000000
F32	abcdefgh	aBbbbbbc defgh000 00000000 00000000
F64	abcdefgh	aBbbbbbb bbcdefgh 00000000 00000000 00000000 00000000 00000000 00000000

a. In this column, B = NOT(b). The bit pattern represents the floating-point number $(-1)^S \times 2^{exp} \times$ mantissa, where S = UInt(a), exp = UInt(NOT(b):c:d)-3 and mantissa = (16+UInt(e:f:g:h))/16.

The immediate value shown in the table is either:

- The value of the imm8 field for an FMOV (scalar, immediate) instruction, see FMOV (scalar, immediate).
- The value obtained by concatenating the a:b:c:d:e:f:g:h fields for an FMOV (vector, immediate) instruction, see FMOV (vector, immediate).

Table C2-2 shows the floating-point constant values encoded in the b:c:d:e:f:g:h fields of the FMOV (vector, immediate) instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C2-2 Floating-point constant values

efgh	bcd							
	000	001	010	011	100	101	110	111
0000	2.0	4.0	8.0	16.0	0.125	0.25	0.5	1.0
0001	2.125	4.25	8.5	17.0	0.1328125	0.265625	0.53125	1.0625
0010	2.25	4.5	9.0	18.0	0.140625	0.28125	0.5625	1.125
0011	2.375	4.75	9.5	19.0	0.1484375	0.296875	0.59375	1.1875
0100	2.5	5.0	10.0	20.0	0.15625	0.3125	0.625	1.25
0101	2.625	5.25	10.5	21.0	0.1640625	0.328125	0.65625	1.3125
0110	2.75	5.5	11.0	22.0	0.171875	0.34375	0.6875	1.375
0111	2.875	5.75	11.5	23.0	0.1796875	0.359375	0.71875	1.4375
1000	3.0	6.0	12.0	24.0	0.1875	0.375	0.75	1.5
1001	3.125	6.25	12.5	25.0	0.1953125	0.390625	0.78125	1.5625
1010	3.25	6.5	13.0	26.0	0.203125	0.40625	0.8125	1.625
1011	3.375	6.75	13.5	27.0	0.2109375	0.421875	0.84375	1.6875
1100	3.5	7.0	14.0	28.0	0.21875	0.4375	0.875	1.75
1101	3.625	7.25	14.5	29.0	0.2265625	0.453125	0.90625	1.8125
1110	3.75	7.5	15.0	30.0	0.234375	0.46875	0.9375	1.875
1111	3.875	7.75	15.5	31.0	0.2421875	0.484375	0.96875	1.9375

C2.2.3.1 Operation of modified immediate constants, floating-point instructions

For an A64 floating-point instruction that uses a modified immediate constant, the operation described by the `VFPExpandImm()` pseudocode function returns the value of the immediate constant.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter C3
A64 Instruction Set Overview

This chapter provides an overview of the A64 instruction set. It contains the following sections:

- Branches, Exception generating, and System instructions.
- Loads and stores.
- Data processing - immediate.
- Data processing - register.
- Data processing - SIMD and floating-point.

For a structured breakdown of instruction groups by encoding, see A64 Instruction Set Encoding.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.1 Branches, Exception generating, and System instructions

This section describes the branch, exception generating, and System instructions. It contains the following subsections:

- Conditional branch.
- Unconditional branch (immediate).
- Unconditional branch (register).
- Direct and indirect branch.
- Exception generation and return.
- System register instructions.
- System instructions.
- Hint instructions.
- Barriers and CLREX instructions.
- Pointer authentication instructions.

Note

Software must:

- Use only `BLR` or `BL` to perform a nested subroutine call when that subroutine is expected to return to the immediately following instruction, that is, the instruction with the address of the `BLR` or `BL` instruction incremented by four.
- Use only `RET` to perform a subroutine return, when that subroutine is expected to have been entered by a `BL` or `BLR` instruction.
- Use only `B`, `BR`, or the instructions listed in Table C3-1 to perform a control transfer that is not a subroutine call or subroutine return described in this *Note*.

C3.1.1 Conditional branch

Conditional branches change the flow of execution depending on the following:

- The current state of the Condition flags.
- The result of comparison between 0 and a value or a bit in a general purpose register.
- The result of comparison between values in two general purpose registers.
- The result of comparison between an immediate value and a value in a general purpose register.

See Table C1-1 for a list of the Condition codes that can be used for `cond`.

Table C3-1 shows the Conditional branch instructions.

Table C3-1 Conditional branch instructions

Mnemonic	Instruction	Branch offset range from the PC	See
`B.cond`	Branch conditionally	±1MB	B.cond
`CBNZ`	Compare and branch if nonzero	±1MB	CBNZ
`CBZ`	Compare and branch if zero	±1MB	CBZ
`TBNZ`	Test bit and branch if nonzero	±32KB	TBNZ
`TBZ`	Test bit and branch if zero	±32KB	TBZ

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.1.2 Unconditional branch (immediate)

Unconditional branch (immediate) instructions change the flow of execution unconditionally by adding an immediate offset with a range of ±128MB to the value of the program counter that fetched the instruction. The `BL` instruction also writes the address of the sequentially following instruction to general-purpose register, X30.

Table C3-2 shows the Unconditional branch instructions with an immediate branch offset.

Table C3-2 Unconditional branch instructions (immediate)

Mnemonic	Instruction	Immediate branch offset range from the PC	See
`B`	Branch unconditionally	±128MB	B
`BL`	Branch with link	±128MB	BL

C3.1.3 Unconditional branch (register)

Unconditional branch (register) instructions change the flow of execution unconditionally by setting the program counter to the value in a general-purpose register. The `BLR` instruction also writes the address of the sequentially following instruction to general-purpose register X30. The `RET` instruction behaves identically to `BR`, but provides an additional hint to the PE that this is a return from a subroutine. Table C3-3 shows Unconditional branch instructions that jump directly to an address held in a general-purpose register.

Table C3-3 Unconditional branch instructions (register)

Mnemonic	Instruction	See
`BLR`	Branch with link to register	BLR
`BR`	Branch to register	BR
`RET`	Return from subroutine	RET

C3.1.4 Direct and indirect branch

Table C3-4 and Table C3-5 shows the Direct branch and Indirect branch instructions.

Table C3-4 Direct branch instructions

Mnemonic	Instruction	See
`B`	Unconditional branch	B
`B.cond`	Conditional branch	B.cond
`BC.cond`	Branch consistent conditionally	BC.cond
`BL`	Branch with link	BL
`CB<cc>`	Compare register with immediate and branch	CB<cc> (immediate)
	Compare registers and branch	CB<cc> (register)
`CBB<cc>`	Compare bytes and branch	CBB<cc>

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
CBH<cc>	Compare halfwords and branch	CBH<cc>
CBZ	Compare and branch on zero	CBZ
CBNZ	Compare and branch on nonzero	CBNZ
TBZ	Test bit and branch if zero	TBZ
TBNZ	Test bit and branch if nonzero	TBZ

Table C3-5 Indirect branch instructions

Mnemonic	Instruction	See
BLR	Branch with link to register	BLR
BLRAA	Branch with link to register, with pointer authentication	BLRAA, BLRAAZ, BLRAB, BLRABZ
BLRAAZ		
BLRAB		
BLRABZ		
BR	Branch to register	BR
BRAA	Branch to register, with pointer authentication	BRAA, BRAAZ, BRAB, BRABZ
BRAAZ		
BRAB		
BRABZ		
ERET	Exception return	ERET
ERETAA	Exception return, with pointer authentication	ERETAA, ERETAB
ERETAB		
RET	Return from subroutine	RET
RETAA	Return from subroutine, with pointer authentication	RETAA, RETAB
RETAB		
RETAASPPC	Return from subroutine, with enhanced pointer authentication using an immediate offset	RETAASPPC, RETABSPPC
RETABSPPC		
RETAASPPCR	Return from subroutine, with enhanced pointer authentication using a register	RETAASPPCR, RETABSPPCR
RETABSPPCR		

Table C3-6 Direct branch instructions

Mnemonic	Instruction	See
B	Unconditional branch	B
B.cond	Conditional branch	B.cond
BL	Branch with link	BL
CBZ	Compare and branch on zero	CBZ
CBNZ	Compare and branch on nonzero	CBNZ

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
TBZ	Test bit and branch if zero	TBZ
TBNZ	Test bit and branch if nonzero	TBZ

Table C3-7 Indirect branch instructions

Mnemonic	Instruction	See
BLR	Branch with link to register	BLR
BLRAA	Branch with link to register, with pointer authentication	BLRAA, BLRAAZ, BLRAB, BLRABZ
BLRAAZ		
BLRAB		
BLRABZ		
BR	Branch to register	BR
BRAA	Branch to register, with pointer authentication	BRAA, BRAAZ, BRAB, BRABZ
BRAAZ		
BRAB		
BRABZ		
ERET	Exception return	ERET
ERETAA	Exception return, with pointer authentication	ERETAA, ERETAB
ERETAB		
RET	Return from subroutine	RET
RETAA	Return from subroutine, with pointer authentication	RETAA, RETAB
RETAB		

C3.1.5 Exception generation and return

This section describes the following exceptions:

- Exception generating.
- Exception return.
- Debug state.

C3.1.5.1 Exception generating

Table C3-8 shows the Exception generating instructions.

Table C3-8 Exception generating instructions

Mnemonic	Instruction	See
BRK	Breakpoint Instruction	BRK

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
HLT	Halt Instruction	HLT
HVC	Generate exception targeting Exception level 2	HVC
SMC	Generate exception targeting Exception level 3	SMC
SVC	Generate exception targeting Exception level 1	SVC

C3.1.5.2 Exception return

Table C3-9 shows the Exception return instructions.

Table C3-9 Exception return instructions

Mnemonic	Instruction	See
ERET	Exception return using current ELR and SPSR	ERET
ERETAA, ERETAB	Exception return, with pointer authentication	ERETAA, ERETAB

C3.1.5.3 Debug state

Table C3-10 shows the Debug state instructions.

Table C3-10 Debug state instructions

Mnemonic	Instruction	See
DCPS1	Debug switch to Exception level 1	DCPS1
DCPS2	Debug switch to Exception level 2	DCPS2
DCPS3	Debug switch to Exception level 3	DCPS3
DRPS	Debug restore PE state	DRPS

C3.1.6 System register instructions

For detailed information about the System register instructions, see The A64 System Instruction Class. Table C3-11 shows the System register instructions.

Table C3-11 System register instructions

Mnemonic	Instruction	See
MRS	Move System register to general-purpose register	MRS
MSR	Move general-purpose register to System register	MSR (register)
	Move immediate to PE state field	MSR (immediate)

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.1.7 System instructions

For detailed information about the System instructions, see The A64 System Instruction Class.

Table C3-12 shows the System instructions.

Table C3-12 System instructions

Mnemonic	Instruction	See
SYS	System instruction	SYS
SYSL	System instruction with result	SYSL
IC	Instruction cache maintenance	IC and Table C5-1
DC	Data cache maintenance	DC and Table C5-1
AT	Address translation	AT and Table C5-2
TLBI	TLB Invalidate	TLBI and Table C5-3

C3.1.8 Hint instructions

Table C3-13 shows the Hint instructions.

Table C3-13 Hint instructions

Mnemonic	Instruction	See
NOP	No operation	NOP
YIELD	Yield hint	YIELD
WFE	Wait for event	WFE
WFI	Wait for interrupt	WFI
SEV	Send event	SEV
SEVL	Send event local	SEVL
HINT	Unallocated hint	HINT
DGH	Data Gathering Hint	DGH

C3.1.9 Barriers and CLREX instructions

Table C3-14 shows the barrier and CLREX instructions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-14 Barriers and CLREX instructions

Mnemonic	Instruction	See
CLREX	Clear Exclusives monitor	CLREX
DFB	Data full barrier	DFB
DMB	Data memory barrier	DMB
DSB	Data synchronization barrier	DSB
ISB	Instruction synchronization barrier	ISB

For more information about DSB, DMB, and ISB, see Memory barriers.

Table C3-15 shows the speculation and synchronization barriers. If these instructions are not implemented, then these instructions execute as a NOP.

Table C3-15 Speculation and synchronization barriers

Mnemonic	Instruction	See
CSDB	Consumption of Speculative Data Barrier	CSDB
ESB	Error synchronization barrier	ESB
PSSBB	Physical Speculative Store Bypass Barrier	PSSBB
SB	Speculation Barrier	SB
SSBB	Speculative Store Bypass Barrier	SSBB
TSB	Trace Synchronization Barrier	TSB

For more information about:

- CSDB, PSSBB, SB, SSBB, TSB, see Memory barriers.
- ESB, see Error synchronization event.

C3.1.10 Pointer authentication instructions

FEAT_PAuth adds support for pointer authentication, see Pointer authentication. This functionality includes the A64 instructions described in this section. These instructions fall into two groups, see:

- Basic pointer authentication instructions.
- Combined instructions that include pointer authentication.

C3.1.10.1 Basic pointer authentication instructions

Each of these instructions only performs an operation that supports pointer authentication.

Table C3-17 shows the instructions that add a *Pointer Authentication Code* (PAC) to the address in a register:

Table C3-16 shows the instructions that authenticate a PAC in a register:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-16 Instructions that authenticate a PAC

Mnemonic	Instruction	See
AUTIASP	Authenticate PAC for instruction address using APIAKey_EL1 and SP	AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA
AUTIAZ	Authenticate PAC for instruction address using APIAKey_EL1 and zero	
AUTIA1716	Authenticate PAC for instruction address X17 using APIAKey_EL1 and X16	
AUTIA171615	Authenticate PAC for instruction address X17 using APIAKey_EL1, X16, and X15	AUTIA171615
AUTIASPPC	Authenticate PAC for instruction address LR using APIAKey_EL1, SP, and PC with an immediate offset	AUTIASPPC
AUTIASPPCR	Authenticate PAC for instruction address LR using APIAKey_EL1, SP, and a register	AUTIASPPCR
AUTIBSP	Authenticate PAC for instruction address using APIBKey_EL1 and SP	AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB
AUTIBZ	Authenticate PAC for instruction address using APIBKey_EL1 and zero	
AUTIB1716	Authenticate PAC for instruction address X17 using APIBKey_EL1 and X16	
AUTIB171615	Authenticate PAC for instruction address X17 using APIBKey_EL1, X16, X15	AUTIB171615
AUTIBSPPC	Authenticate PAC for instruction address LR using APIBKey_EL1, SP, and PC with an immediate offset	AUTIBSPPC
AUTIBSPPCR	Authenticate PAC for instruction address LR using APIBKey_EL1, SP, and a register	AUTIBSPPCR
AUTIA	Authenticate PAC for instruction address using APIAKey_EL1, registers	AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA
AUTDA	Authenticate PAC for data address using APDAKey_EL1, registers	AUTDA, AUTDZA
AUTIB	Authenticate PAC for instruction address using APIBKey_EL1, registers	AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB
AUTDB	Authenticate PAC for data address using APDBKey_EL1, registers	AUTDB, AUTDZB
AUTIZA	Authenticate PAC for instruction address using APIAKey_EL1, register and zero	AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA
AUTDZA	Authenticate PAC for data address using APDAKey_EL1, register and zero	AUTDA, AUTDZA
AUTIZB	Authenticate PAC for instruction address using APIBKey_EL1, register and zero	AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB
AUTDZB	Authenticate PAC for data address using APDBKey_EL1, register and zero	AUTDB, AUTDZB

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-17 Instructions that add a PAC

Mnemonic	Instruction	See
PACIASP	Add PAC to instruction address using APIAKey_EL1 and SP	PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA
PACIAZ	Add PAC to instruction address using APIAKey_EL1 and zero	
PACIA1716	Add PAC to instruction address X17 using APIAKey_EL1 and X16	
PACIBSP	Add PAC to instruction address using APIBKey_EL1 and SP	PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB
PACIBZ	Add PAC to instruction address using APIBKey_EL1 and zero	
PACIB1716	Add PAC to instruction address X17 using APIBKey_EL1 and X16	
PACIA	Add PAC to instruction address using APIAKey_EL1, registers	PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA
PACDA	Add PAC to data address using APDAKey_EL1, registers	PACDA, PACDZA
PACIB	Add PAC to instruction address using APIBKey_EL1, registers	PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB
PACDB	Add PAC to data address using APDBKey_EL1, registers	PACDB, PACDZB
PACIZA	Add PAC to instruction address using APIAKey_EL1, register and zero	PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA
PACDZA	Add PAC to data address using APDAKey_EL1, register and zero	PACDA, PACDZA
PACIZB	Add PAC to instruction address using APIBKey_EL1, register and zero	PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB
PACDZB	Add PAC to data address using APDBKey_EL1, register and zero	PACDB, PACDZB
PACGA	Add generic PAC using APGAKey_EL1, registers	PACGA

Table C3-16 shows the instructions that authenticate a PAC in a register:

Table C3-18 Instructions that authenticate a PAC

Mnemonic	Instruction	See
AUTIASP	Authenticate PAC for instruction address using APIAKey_EL1 and SP	AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA
AUTIAZ	Authenticate PAC for instruction address using APIAKey_EL1 and zero	
AUTIA1716	Authenticate PAC for instruction address X17 using APIAKey_EL1 and X16	
AUTIBSP	Authenticate PAC for instruction address using APIBKey_EL1 and SP	AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB
AUTIBZ	Authenticate PAC for instruction address using APIBKey_EL1 and zero	

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
AUTIB1716	Authenticate PAC for instruction address X17 using APIBKey_EL1 and X16	
AUTIA	Authenticate PAC for instruction address using APIAKey_EL1, registers	AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA
AUTDA	Authenticate PAC for data address using APDAKey_EL1, registers	AUTDA, AUTDZA
AUTIB	Authenticate PAC for instruction address using APIBKey_EL1, registers	AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB
AUTDB	Authenticate PAC for data address using APDBKey_EL1, registers	AUTDB, AUTDZB
AUTIZA	Authenticate PAC for instruction address using APIAKey_EL1, register and zero	AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA
AUTDZA	Authenticate PAC for data address using APDAKey_EL1, register and zero	AUTDA, AUTDZA
AUTIZB	Authenticate PAC for instruction address using APIBKey_EL1, register and zero	AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB
AUTDZB	Authenticate PAC for data address using APDBKey_EL1, register and zero	AUTDB, AUTDZB

Table C3-19 shows the instructions that strip a PAC from a register, without performing any authentication:

Table C3-19 Instructions that strip a PAC

Mnemonic	Instruction	See
XPACLRI	Strip instruction address PAC from LR	XPACD, XPACI, XPACLRI
XPACI	Strip instruction address PAC, register	
XPACD	Strip data address PAC, register	

C3.1.10.2 Combined instructions that include pointer authentication

Each of these instructions combines a pointer authentication with another operation that uses the authenticated pointer. Table C3-20 shows these instructions:

Table C3-20 Combined pointer authentication instructions

Mnemonic	Instruction	See
RETAA	Authenticate PAC for LR using APIAKey_EL1 and SP, and return	RETAA, RETAB
RETAB	Authenticate PAC for LR using APIBKey_EL1 and SP, and return	
BRAA	Authenticate PAC using APIAKey_EL1 (registers), and branch	BRAA, BRAAZ, BRAB, BRABZ
BRAB	Authenticate PAC using APIBKey_EL1 (registers), and branch	
BLRAA	Authenticate PAC using APIAKey_EL1 (registers), and branch with link	BLRAA, BLRAAZ, BLRAB, BLRABZ

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
BLRAB	Authenticate PAC using APIBKey_EL1 (registers), and branch with link	
BRAAZ	Authenticate PAC using APIAKey_EL1 (register and zero), and branch	BRAA, BRAAZ, BRAB, BRABZ
BRABZ	Authenticate PAC using APIBKey_EL1 (register and zero), and branch	
BLRAAZ	Authenticate PAC using APIAKey_EL1 (register and zero), and branch with link	BLRAA, BLRAAZ, BLRAB, BLRABZ
BLRABZ	Authenticate PAC using APIBKey_EL1 (register and zero), and branch with link	
ERETAA	Authenticate PAC for ELR using APIAKey_EL1 and SP, and exception return	ERETAA, ERETAB
ERETAB	Authenticate PAC for ELR using APIBKey_EL1 and SP, and exception return	
LDRAA	Authenticate PAC for data address using APDAKey_EL1 (register and zero) and Load	LDRAA, LDRAB
LDRAB	Authenticate PAC for data address using APDBKey_EL1 (register and zero) and Load	

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.2 Loads and stores

This section describes the load/store instructions. It contains the following subsections:

- Load/store register.
- Load/store register (unscaled offset).
- Load/store pair.
- Load/store non-temporal pair.
- Load/store unprivileged.
- Load-Exclusive/Store-Exclusive.
- Load-Acquire/Store-Release.
- Load/store scalar SIMD and floating-point.
- Load/store Advanced SIMD.
- Prefetch memory.
- Atomic instructions.

The requirements for the alignment of data memory accesses are strict. For more information, see Alignment of data accesses.

The additional control bits SCTLR_ELx.SA and SCTLR_EL1.SA0 control whether the stack pointer must be quadword aligned when used as a base register. See SP alignment checking. Using a misaligned stack pointer generates an SP alignment fault exception.

Note

In some cases, load/store instructions can lead to CONSTRAINED UNPREDICTABLE behavior. See AArch64 CONSTRAINED UNPREDICTABLE behaviors.

C3.2.1 Load/store register

The load/store register instructions support the following addressing modes:

- Base plus a scaled 12-bit unsigned immediate offset or base plus an unscaled 9-bit signed immediate offset.
- Base plus a 64-bit register offset, optionally scaled.
- Base plus a 32-bit extended register offset, optionally scaled.
- Pre-indexed by an unscaled 9-bit signed immediate offset.
- Post-indexed by an unscaled 9-bit signed immediate offset.
- PC-relative literal for loads of 32 bits or more.

See also Load/store addressing modes.

If a Load instruction specifies writeback and the register being loaded is also the base register, then behavior is CONSTRAINED UNPREDICTABLE and one of the following behaviors must occur:

- The instruction is treated as UNDEFINED.
- The instruction is treated as a NOP.
- The instruction performs the load using the specified addressing mode and the base register becomes UNKNOWN. In addition, if an exception occurs during the execution of such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

If a Store instruction performs a writeback and the register that is stored is also the base register, then behavior is CONSTRAINED UNPREDICTABLE and one of the following behaviors must occur:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The instruction is treated as UNDEFINED.
- The instruction is treated as a NOP.
- The instruction performs the store to the designated register using the specified addressing mode, but the value stored is UNKNOWN.

Table C3-21 shows the load/store register instructions.

Table C3-21 Load/store register instructions

Mnemonic	Instruction	See
LDR	Load register (register offset)	LDR (register)
	Load register (immediate offset)	LDR (immediate)
	Load register (PC-relative literal)	LDR (literal)
LDRB	Load byte (register offset)	LDRB (register)
	Load byte (immediate offset)	LDRB (immediate)
LDRSB	Load signed byte (register offset)	LDRSB (register)
	Load signed byte (immediate offset)	LDRSB (immediate)
LDRH	Load halfword (register offset)	LDRH (register)
	Load halfword (immediate offset)	LDRH (immediate)
LDRSH	Load signed halfword (register offset)	LDRSH (register)
	Load signed halfword (immediate offset)	LDRSH (immediate)
LDRSW	Load signed word (register offset)	LDRSW (register)
	Load signed word (immediate offset)	LDRSW (immediate)
	Load signed word (PC-relative literal)	LDRSW (literal)
STR	Store register (register offset)	STR (register)
	Store register (immediate offset)	STR (immediate)
STRB	Store byte (register offset)	STRB (register)
	Store byte (immediate offset)	STRB (immediate)
STRH	Store halfword (register offset)	STRH (register)
	Store halfword (immediate offset)	STRH (immediate)

C3.2.2 Load/store register (unscaled offset)

The load/store register instructions with an unscaled offset support only one addressing mode:

- Base plus an unscaled 9-bit signed immediate offset.

See Load/store addressing modes.

The load/store register (unscaled offset) instructions are required to disambiguate this instruction class from the load/store register instruction forms that support an addressing mode of base plus a scaled, unsigned 12-bit immediate offset, because that can represent some offset values in the same range.

The ambiguous immediate offsets are byte offsets that are both:

- In the range 0-255, inclusive.
- Naturally aligned to the access size.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Other byte offsets in the range -256 to 255 inclusive are unambiguous. An assembler program translating a load/store instruction, for example `LDR`, is required to encode an unambiguous offset using the unscaled 9-bit offset form, and to encode an ambiguous offset using the scaled 12-bit offset form. A programmer might force the generation of the unscaled 9-bit form by using one of the mnemonics in Table C3-22. Arm recommends that a disassembler outputs all unscaled 9-bit offset forms using one of these mnemonics, but unambiguous offsets can be output using a load/store single register mnemonic, for example, `LDR`.

Table C3-22 shows the load/store register instructions with an unscaled offset.

Table C3-22 Load/store register (unscaled offset) instructions

Mnemonic	Instruction	See
`LDUR`	Load register (unscaled offset)	LDUR
`LDURB`	Load byte (unscaled offset)	LDURB
`LDURSB`	Load signed byte (unscaled offset)	LDURSB
`LDURH`	Load halfword (unscaled offset)	LDURH
`LDURSH`	Load signed halfword (unscaled offset)	LDURSH
`LDURSW`	Load signed word (unscaled offset)	LDURSW
`STUR`	Store register (unscaled offset)	STUR
`STURB`	Store byte (unscaled offset)	STURB
`STURH`	Store halfword (unscaled offset)	STURH

C3.2.3 Load/store pair

The load/store pair instructions support the following addressing modes:

- Base plus a scaled 7-bit signed immediate offset.
- Pre-indexed by a scaled 7-bit signed immediate offset.
- Post-indexed by a scaled 7-bit signed immediate offset.

See also Load/store addressing modes.

If a Load Pair instruction specifies the same register for the two registers that are being loaded, then behavior is CONSTRAINED UNPREDICTABLE and one of the following behaviors must occur:

- The instruction is treated as UNDEFINED.
- The instruction is treated as a `NOP`.
- The instruction performs all the loads using the specified addressing mode and the register that is loaded takes an UNKNOWN value.

If a Load Pair instruction specifies writeback and one of the registers being loaded is also the base register, then behavior is CONSTRAINED UNPREDICTABLE and one of the following behaviors must occur:

- The instruction is treated as UNDEFINED.
- The instruction is treated as a `NOP`.
- The instruction performs all of the loads using the specified addressing mode, and the base register becomes UNKNOWN. In addition, if an exception occurs during the instruction, the base address might be corrupted so that the instruction cannot be repeated.

If a Store Pair instruction performs a writeback and one of the registers being stored is also the base register, then behavior is CONSTRAINED UNPREDICTABLE and one of the following behaviors must occur:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The instruction is treated as UNDEFINED.
- The instruction is treated as a NOP.
- The instruction performs all the stores of the registers indicated by the specified addressing mode, but the value stored for the base register is UNKNOWN.

Table C3-23 shows the load/store pair instructions.

Table C3-23 Load/store pair instructions

Mnemonic	Instruction	See
LDP	Load Pair	LDP
LDPSW	Load Pair signed words	LDPSW
STP	Store Pair	STP

C3.2.4 Load/store non-temporal pair

The load/store non-temporal pair instructions support only one addressing mode:

- Base plus a scaled 7-bit signed immediate offset.

See Load/store addressing modes.

The load/store non-temporal pair instructions provide a hint to the memory system that an access is non-temporal, and unlikely to be repeated in the near future. This means that data caching is not required. However, depending on the memory type, the instructions might permit memory reads to be prefetched and memory writes to be gathered to accelerate bulk memory transfers.

In addition, there is an exception to the usual memory ordering rules. If an address dependency exists between two memory reads, and a Load Non-temporal Pair instruction generated the second read, then in the absence of any other barrier mechanism to achieve order, the memory accesses can be observed in any order by the other observers within the shareability domain of the memory addresses being accessed.

If a Load Non-Temporal Pair instruction specifies the same register for the two registers that are being loaded, then behavior is CONSTRAINED UNPREDICTABLE and one of the following must occur:

- The instruction is treated as UNDEFINED.
- The instruction is treated as a NOP.
- The instruction performs all the loads using the specified addressing mode and the register that is loaded takes an UNKNOWN value.

Table C3-24 shows the load/store non-temporal pair instructions.

Table C3-24 Load/store non-temporal pair instructions

Mnemonic	Instruction	See
LDNP	Load Non-temporal Pair	LDNP
STNP	Store Non-temporal Pair	STNP

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.2.5 Load/store unprivileged

The load/store unprivileged instructions support only one addressing mode:

- Base plus an unscaled 9-bit signed immediate offset.

See Load/store addressing modes.

The access permissions that apply to accesses made at EL0 apply to the memory accesses made by a load/store unprivileged instruction that is executed either:

- At EL1 when the Effective value of PSTATE.UAO is 0.
- At EL2 when both the Effective value of HCR_EL2.{E2H, TGE} is {1, 1} and the Effective value of PSTATE.UAO is 0.

Otherwise, memory accesses made by a load/store unprivileged instruction are subject to the access permissions that apply to the Exception level at which the instruction is executed. These are the permissions that apply to the corresponding load/store register instruction, see Load/store register.

Note

This means that when the value of PSTATE.UAO is 1 the access permissions for a load/store unprivileged instruction are always the same as those for the corresponding load/store register instruction.

Table C3-25 shows the load/store unprivileged instructions.

Table C3-25 Load-Store unprivileged instructions

Mnemonic	Instruction	See
LDTR	Load unprivileged register	LDTR
LDTRB	Load unprivileged byte	LDTRB
LDTRSB	Load unprivileged signed byte	LDTRSB
LDTRH	Load unprivileged halfword	LDTRH
LDTRSH	Load unprivileged signed halfword	LDTRSH
LDTRSW	Load unprivileged signed word	LDTRSW
STTR	Store unprivileged register	STTR
STTRB	Store unprivileged byte	STTRB
STTRH	Store unprivileged halfword	STTRH

C3.2.6 Load-Exclusive/Store-Exclusive

The Load-Exclusive/Store-Exclusive instructions support only one addressing mode:

- Base register with no offset.

See Load/store addressing modes.

The Load-Exclusive instructions mark the physical address being accessed as an exclusive access. This exclusive access mark is checked by the Store-Exclusive instruction, permitting the construction of atomic read-modify-write operations on shared memory variables, semaphores, mutexes, and spinlocks. See Synchronization and semaphores.

If FEAT_LSE2 is not implemented then:.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The Load-Exclusive/Store-Exclusive instructions other than Load-Exclusive pair and Store-Exclusive pair require natural alignment, and an unaligned address generates an Alignment fault.
- Memory accesses generated by Load-Exclusive pair or Store-Exclusive pair instructions must be aligned to the size of the pair, otherwise the access generates an Alignment fault.

  For more information on alignment requirements and behaviors, see Load-Exclusive/ Store-Exclusive and Atomic instructions.

When a Store-Exclusive pair succeeds, it causes a single-copy atomic update of the entire memory location being stored to.

Table C3-26 shows the Load-Exclusive/Store-Exclusive instructions.

Table C3-26 Load-Exclusive/Store-Exclusive instructions

Mnemonic	Instruction	See
LDXR	Load Exclusive register	LDXR
LDXRB	Load Exclusive byte	LDXRB
LDXRH	Load Exclusive halfword	LDXRH
LDXP	Load Exclusive pair	LDXP
STXR	Store Exclusive register	STXR
STXRB	Store Exclusive byte	STXRB
STXRH	Store Exclusive halfword	STXRH
STXP	Store Exclusive pair	STXP

C3.2.7 Load-Acquire/Store-Release

The Load-Acquire, Load-AcquirePC and Store-Release instructions are added as part of the FEAT_LRCPC, FEAT_LRCPC2, and FEAT_LRCPC3 architecture features.

If FEAT_LSE2 is not implemented, then:

- The Load-Acquire, Load-AcquirePC, and Store-Release instructions other than Load-Acquire pair and Store-Release pair require natural alignment, and an unaligned address generates an Alignment fault.
- Memory accesses generated by Load-Acquire pair or Store-Release pair instructions must be aligned to the size of the accessed element in the pair, otherwise the access generates an Alignment fault.

For more information on alignment requirements and behaviors, see Non-atomic Load-Acquire/Store-Release instructions.

The Load-Acquire, Load-AcquirePC, and Store-Release instructions can remove the requirement to use the explicit DMB memory barrier instruction. For more information about the ordering of Load-Acquire, Load-AcquirePC, and Store-Release, see Load-Acquire, Load-AcquirePC, and Store-Release.

A Store-Release Exclusive instruction has the Release semantics only if the store is successful.

Armv8.1 adds more instructions with load-acquire and store-release mechanisms, see LoadLOAcquire/StoreLORelease.

Table C3-27 shows the Non-exclusive Load-Acquire/Store-Release instructions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-27 Non-exclusive Load-Acquire and Store-Release instructions

Mnemonic	Instruction	See
LDAPR	Load-Acquire RCpc Register	LDAPR
LDAPRB	Load-Acquire RCpc Register Byte	LDAPRB
LDAPRH	Load-Acquire RCpc Register Halfword	LDAPRH
LDAPUR	Load-Acquire RCpc Register (unscaled)	LDAPUR
LDAPURB	Load-Acquire RCpc Register Byte (unscaled)	LDAPURB
LDAPURH	Load-Acquire RCpc Register Halfword (unscaled)	LDAPURH
LDAPURSB	Load-Acquire RCpc Register Signed Byte (unscaled) 32-bit	LDAPURSB
LDAPURSB	Load-Acquire RCpc Register Signed Byte (unscaled) 64-bit	LDAPURSB
LDAPURSH	Load-Acquire RCpc Register Signed Halfword (unscaled) 32-bit	LDAPURSH
LDAPURSH	Load-Acquire RCpc Register Signed Halfword (unscaled) 64-bit	LDAPURSH
LDAPURSW	Load-Acquire RCpc Register Signed Word (unscaled)	LDAPURSW
LDAR	Load-Acquire Register	LDAR
LDARB	Load-Acquire Byte	LDARB
LDARH	Load-Acquire Halfword	LDARH
STLR	Store-Release Register	STLR
STLRB	Store-Release Byte	STLRB
STLRH	Store-Release Halfword	STLRH
STLUR	Store-Release Register (unscaled)	STLUR
STLURB	Store-Release Register Byte (unscaled)	STLURB
STLURH	Store-Release Register Halfword (unscaled)	STLURH

Table C3-28 shows the Exclusive Load-Acquire/Store-Release instructions.

Table C3-28 Exclusive Load-Acquire and Store-Release instructions

Mnemonic	Instruction	See
LDAXR	Load-Acquire Exclusive register	LDAXR
LDAXRB	Load-Acquire Exclusive byte	LDAXRB
LDAXRH	Load-Acquire Exclusive halfword	LDAXRH
LDAXP	Load-Acquire Exclusive pair	LDAXP
STLXR	Store-Release Exclusive register	STLXR
STLXRB	Store-Release Exclusive byte	STLXRB
STLXRH	Store-Release Exclusive halfword	STLXRH
STLXP	Store-Release Exclusive pair	STLXP

C3.2.8 Load/store scalar SIMD and floating-point

The load/store scalar SIMD and floating-point instructions operate on scalar values in the SIMD and floating-point register file as described in Advanced SIMD and floating-point scalar register names. The memory addressing modes

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

available, described in Load/store addressing modes, are identical to the general-purpose register load/store instructions, and like those instructions permit arbitrary address alignment unless strict alignment checking is enabled. However, unlike the load/store instructions that transfer general-purpose registers, load/store scalar SIMD and floating-point instructions make no guarantee of atomicity, even when the address is naturally aligned to the size of the data.

C3.2.8.1 Load/store scalar SIMD and floating-point register

The load/store scalar SIMD and floating-point register instructions support the following addressing modes:

- Base plus a scaled 12-bit unsigned immediate offset or base plus unscaled 9-bit signed immediate offset.
- Base plus 64-bit register offset, optionally scaled.
- Base plus 32-bit extended register offset, optionally scaled.
- Pre-indexed by an unscaled 9-bit signed immediate offset.
- Post-indexed by an unscaled 9-bit signed immediate offset.
- PC-relative literal for loads of 32 bits or more.

For more information on the addressing modes, see Load/store addressing modes.

Note

The unscaled 9-bit signed immediate offset address mode requires its own instruction form, see Load/store scalar SIMD and floating-point register (unscaled offset).

Table C3-29 shows the load/store instructions for a single SIMD and floating-point register.

Table C3-29 Load/store single SIMD and floating-point register instructions

Mnemonic	Instruction	See
LDR	Load scalar SIMD&FP register (register offset)	LDR (register, SIMD&FP)
	Load scalar SIMD&FP register (immediate offset)	LDR (immediate, SIMD&FP)
	Load scalar SIMD&FP register (PC-relative literal)	LDR (literal, SIMD&FP)
STR	Store scalar SIMD&FP register (register offset)	STR (register, SIMD&FP)
	Store scalar SIMD&FP register (immediate offset)	STR (immediate, SIMD&FP)

C3.2.8.2 Load/store scalar SIMD and floating-point register (unscaled offset)

The load /store scalar SIMD and floating-point register instructions support only one addressing mode:

- Base plus an unscaled 9-bit signed immediate offset.

See also Load/store addressing modes.

The load/store scalar SIMD and floating-point register (unscaled offset) instructions are required to disambiguate this instruction class from the load/store single SIMD and floating-point instruction forms that support an addressing mode of base plus a scaled, unsigned 12-bit immediate offset. This is similar to the load/store register (unscaled offset) instructions, that disambiguate this instruction class from the load/store register instruction, see Load/store register (unscaled offset).

Table C3-30 shows the load/store SIMD and floating-point register instructions with an unscaled offset.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-30 Load/store SIMD and floating-point register instructions

Mnemonic	Instruction	See
LDUR	Load scalar SIMD&FP register (unscaled offset)	LDUR (SIMD&FP)
STUR	Store scalar SIMD&FP register (unscaled offset)	STUR (SIMD&FP)

C3.2.8.3 Load/store SIMD and floating-point register pair

The load/store SIMD and floating-point register pair instructions support the following addressing modes:

- Base plus a scaled 7-bit signed immediate offset.
- Pre-indexed by a scaled 7-bit signed immediate offset.
- Post-indexed by a scaled 7-bit signed immediate offset.

See also Load/store addressing modes.

If a Load pair instruction specifies the same register for the two registers that are being loaded, then behavior is CONSTRAINED UNPREDICTABLE and one of the following behaviors must occur:

- The instruction is treated as UNDEFINED.
- The instruction is treated as a NOP.
- The instruction performs all of the loads using the specified addressing mode and the register being loaded takes an UNKNOWN value.

Table C3-31 shows the load/store SIMD and floating-point register pair instructions.

Table C3-31 Load/store SIMD and floating-point register pair instructions

Mnemonic	Instruction	See
LDP	Load pair of scalar SIMD&FP registers	LDP (SIMD&FP)
STP	Store pair of scalar SIMD&FP registers	STP (SIMD&FP)

C3.2.8.4 Load/store SIMD and floating-point non-temporal pair

The load/store SIMD and floating-point non-temporal pair instructions support only one addressing mode:

- Base plus a scaled 7-bit signed immediate offset.

See also Load/store addressing modes.

The load/store non-temporal pair instructions provide a hint to the memory system that an access is non-temporal or streaming, and unlikely to be repeated in the near future. This means that data caching is not required. However, depending on the memory type, the instructions might permit memory reads to be prefetched and memory writes to be gathered to accelerate bulk memory transfers.

In addition, there is an exception to the usual memory ordering rules. If an address dependency exists between two memory reads, and a load non-temporal pair instruction generated the second read, then in the absence of any other barrier mechanism to achieve order, those memory accesses can be observed in any order by the other observers within the shareability domain of the memory addresses being accessed.

If a load non-temporal pair instruction specifies the same register for the two registers that are being loaded, then behavior is CONSTRAINED UNPREDICTABLE and one of the following behaviors must occur:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The instruction is treated as UNDEFINED.
- The instruction is treated as a NOP.
- The instruction performs all the loads using the specified addressing mode and the register that is loaded takes an UNKNOWN value.

Table C3-32 shows the load/store SIMD and floating-point Non-temporal pair instructions.

Table C3-32 Load/store SIMD and floating-point non-temporal pair instructions

Mnemonic	Instruction	See
LDNP	Load pair of scalar SIMD&FP registers	LDNP (SIMD&FP)
STNP	Store pair of scalar SIMD&FP registers	STNP (SIMD&FP)

C3.2.9 Load/store Advanced SIMD

The Advanced SIMD load/store structure instructions support the following addressing modes:

- Base register only.
- Post-indexed by a 64-bit register.
- Post-indexed by an immediate, equal to the number of bytes transferred.

Load/store vector instructions, like other load/store instructions, allow any address alignment, unless strict alignment checking is enabled. If strict alignment checking is enabled, then alignment checking to the size of the element is performed. However, unlike the load/store instructions that transfer general-purpose registers, the load/store vector instructions do not guarantee atomicity, even when the address is naturally aligned to the size of the element.

C3.2.9.1 Load/store structures

Table C3-33 shows the load/store structure instructions. A post-increment immediate offset, if present, must be 8, 16, 24, 32, 48, or 64, depending on the number of elements transferred.

Table C3-33 Load/store multiple structures instructions

Mnemonic	Instruction	See
LD1	Load single 1-element structure to one lane of one register	LD1 (single structure)
	Load multiple 1-element structures to one register or to two, three, or four consecutive registers	LD1 (multiple structures)
LD2	Load single 2-element structure to one lane of two consecutive registers	LD2 (single structure)
	Load multiple 2-element structures to two consecutive registers	LD2 (multiple structures)
LD3	Load single 3-element structure to one lane of three consecutive registers	LD3 (single structure)
	Load multiple 3-element structures to three consecutive registers	LD3 (multiple structures)
LD4	Load single 4-element structure to one lane of four consecutive registers	LD4 (single structure)
	Load multiple 4-element structures to four consecutive registers	LD4 (multiple structures)
ST1	Store single 1-element structure from one lane of one register	ST1 (single structure)

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
	Store multiple 1-element structures from one register, or from two, three, or four consecutive registers	ST1 (multiple structures)
ST2	Store single 2-element structure from one lane of two consecutive registers	ST2 (single structure)
	Store multiple 2-element structures from two consecutive registers	ST2 (multiple structures)
ST3	Store single 3-element structure from one lane of three consecutive registers	ST3 (single structure)
	Store multiple 3-element structures from three consecutive registers	ST3 (multiple structures)
ST4	Store single 4-element structure from one lane of four consecutive registers	ST4 (single structure)
	Store multiple 4-element structures from four consecutive registers	ST4 (multiple structures)

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.2.9.2 Load single structure and replicate

Table C3-34 shows the Load single structure and replicate instructions. A post-increment immediate offset, if present, must be 1, 2, 3, 4, 6, 8, 12, 16, 24, or 32, depending on the number of elements transferred.

Table C3-34 Load single structure and replicate instructions

Mnemonic	Instruction	See
LD1R	Load single 1-element structure and replicate to all lanes of one register	LD1R
LD2R	Load single 2-element structure and replicate to all lanes of two registers	LD2R
LD3R	Load single 3-element structure and replicate to all lanes of three registers	LD3R
LD4R	Load single 4-element structure and replicate to all lanes of four registers	LD4R

C3.2.10 Prefetch memory

The Prefetch memory instructions support the following addressing modes:

- Base register with no offset.
- Base plus a scaled 12-bit unsigned immediate offset or base plus an unscaled 9-bit signed immediate offset.
- Base plus a 64-bit register offset. This can be optionally scaled by 8-bits, for example LSL#3.
- Base plus a 32-bit extended register offset. This can be optionally scaled by 8-bits.
- PC-relative literal.

The Prefetch memory instructions signal to the memory system that memory accesses from a specified address or range of addresses are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory access when they do occur, such as prefetching the specified address or range of addresses into one or more caches. Because these signals are only hints, it is valid for the PE to treat any or all prefetch instructions as a NOP.

Because they are hints to the memory system, the operation of a prefetch memory instruction cannot cause a synchronous exception. However, a memory operation performed as a result of one of these memory system hints might in exceptional cases trigger an asynchronous event, and thereby influence the execution of the PE. An example of an asynchronous event that might be triggered is an SError exception.

A prefetch memory instruction can only have an effect on software visible structures as defined in the translation regime of the current Exception level, such as caches and translation lookaside buffers associated with memory locations that can be accessed by reads, writes, or execution.

A PRFM, PRFUM, or RPRFM instruction is guaranteed not to access Device memory.

A PRFM or PRFUM instruction using a PLI hint cannot result in any access that could not be performed by the PE speculatively fetching an instruction. Therefore, if all associated MMUs and MPUs are disabled, a PLI hint can only access memory locations consistent with instruction fetches permitted by The effects of disabling an address translation stage.

Arm recommends that when a PRFM or a PRFUM instruction describes a particular cache level, then the memory location described at that cache level should be made available at that same cache level, even if the line had previously been at a level of the cache that is closer to the PE than the specified cache level.

Note

For a PRFM PST instruction, this might result in the demotion of a cache line from a level that was closer to the cache.

Table C3-35 shows the Prefetch memory instructions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-35 Prefetch memory instructions

Mnemonic	Instruction	See
PRFM	Prefetch memory (register offset)	PRFM (register)
	Prefetch memory (immediate offset)	PRFM (immediate)
	Prefetch memory (PC-relative offset)	PRFM (literal)
PRFUM	Prefetch memory (unscaled offset)	PRFUM

C3.2.11 Atomic instructions

The atomic instructions perform atomic read and write operations on a memory location such that the architecture guarantees that no modification of that memory location by another observer can occur between the read and the write defined by that instruction.

This section describes the following operations:

- Atomic integer memory operations.
- Atomic floating-point memory operations.
- Swap operations.
- Compare and Swap operations.

The atomic instructions support only one addressing mode:

- Base register only.

See also Load/store addressing modes.

For the purpose of permission checking, and for watchpoints, all of the Atomic memory operation instructions are treated as performing both a load and a store.

The instructions are provided with ordering options, which map to the acquire and release definitions used in the architecture. The atomic instructions with release semantics have the same rules as Store-Release instructions regarding multi-copy atomicity. These operations map to the acquire and release definitions, and are counted as Load-Acquire and Store-Release operations respectively.

C3.2.11.1 Atomic integer memory operations

The address accessed by the 128-bit atomic instructions must be aligned to 128-bits, otherwise the instructions generate an Alignment fault.

For the LD<OP> instructions, where the source and destination registers are the same, if the instruction generates a synchronous Data Abort, then the source register is restored to the value it held before the instruction was executed.

The ST<OP> instructions, and LD<OP> instructions where the destination register is WZR or XZR, are not regarded as doing a read for the purpose of a DMB LD barrier.

For information on alignment requirements and behaviors, see Load-Exclusive/ Store-Exclusive and Atomic instructions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-36 Atomic integer memory operation instructions

Mnemonic	Instruction	See
LDADD	Atomic add	LDADD, LDADDA, LDADDAL, LDADDL
LDADDB	Atomic add on byte	LDADDB, LDADDAB, LDADDALB, LDADDLB
LDADDH	Atomic add on halfword	LDADDH, LDADDAH, LDADDALH, LDADDLH
LDCLR	Atomic bit clear	LDCLR, LDCLRA, LDCLRAL, LDCLRL
LDCLRB	Atomic bit clear on byte	LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB
LDCLRH	Atomic bit clear on halfword	LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH
LDEOR	Atomic exclusive-OR	LDEOR, LDEORA, LDEORAL, LDEORL
LDEORB	Atomic exclusive-OR on byte	LDEORB, LDEORAB, LDEORALB, LDEORLB
LDEORH	Atomic exclusive-OR on halfword	LDEORH, LDEORAH, LDEORALH, LDEORLH
LDSET	Atomic bit set	LDSET, LDSETA, LDSETAL, LDSETL
LDSETB	Atomic bit set on byte	LDSETB, LDSETAB, LDSETALB, LDSETLB
LDSETH	Atomic bit set on halfword	LDSETH, LDSETAH, LDSETALH, LDSETLH
LDMAX	Atomic signed maximum	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL
LDMAXB	Atomic signed maximum on byte	LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB
LDMAXH	Atomic signed maximum on halfword	LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH
LDMIN	Atomic signed minimum	LDSMIN, LDSMINA, LDSMINAL, LDSMINL
LDMINB	Atomic signed minimum on byte	LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB
LDMINH	Atomic signed minimum on halfword	LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH
LDUMAX	Atomic unsigned maximum	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL
LDUMAXB	Atomic unsigned maximum on byte	LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB
LDUMAXH	Atomic unsigned maximum on halfword	LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH
LDUMIN	Atomic unsigned minimum	LDUMIN, LDUMINA, LDUMINAL, LDUMINL
LDUMINB	Atomic unsigned minimum on byte	LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB
LDUMINH	Atomic unsigned minimum on halfword	LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH
STADD	Atomic add, without return	STADD, STADDL
STADDB	Atomic add on byte, without return	STADDB, STADDLB
STADDH	Atomic add on halfword, without return	STADDH, STADDLH
STCLR	Atomic bit clear, without return	STCLR, STCLRL
STCLRB	Atomic bit clear on byte, without return	STCLRB, STCLRLB
STCLRH	Atomic bit clear on halfword, without return	STCLRH, STCLRLH
STEOR	Atomic exclusive-OR, without return	STEOR, STEORL
STEORB	Atomic exclusive-OR on byte, without return	STEORB, STEORLB
STEORH	Atomic exclusive-OR on halfword, without return	STEORH, STEORLH
STSET	Atomic bit set, without return	STSET, STSETL
STSETB	Atomic bit set on byte, without return	STSETB, STSETLB
STSETH	Atomic bit set on halfword, without return	STSETH, STSETLH

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
STMAX	Atomic signed maximum, without return	STSMAX, STSMAXL
STMAXB	Atomic signed maximum on byte, without return	STSMAXB, STSMAXLB
STMAXH	Atomic signed maximum on halfword, without return	STSMAXH, STSMAXLH
STMIN	Atomic signed minimum, without return	STSMIN, STSMINL
STMINB	Atomic signed minimum on byte, without return	STSMINB, STSMINLB
STMINH	Atomic signed minimum on halfword, without return	STSMINH, STSMINLH
STUMAX	Atomic unsigned maximum, without return	STUMAX, STUMAXL
STUMAXB	Atomic unsigned maximum on byte, without return	STUMAXB, STUMAXLB
STUMAXH	Atomic unsigned maximum on halfword, without return	STUMAXH, STUMAXLH
STUMIN	Atomic unsigned minimum, without return	STUMIN, STUMINL
STUMINB	Atomic unsigned minimum on byte, without return	STUMINB, STUMINLB
STUMINH	Atomic unsigned minimum on halfword, without return	STUMINH, STUMINLH

C3.2.11.2 Swap operations

For the SWP instructions, where the source and destination registers are the same, if the instruction generates a synchronous Data Abort, then the source register is restored to the value it held before the instruction was executed.

If the destination register Rt is WZR or XZR, this is not regarded as doing a read for the purpose of a DMB LD barrier.

For information on alignment requirements and behaviors, see Load-Exclusive/ Store-Exclusive and Atomic instructions.

Table C3-37 Swap instructions

Mnemonic	Instruction	See
SWP	Swap	SWP, SWPA, SWPAL, SWPL
SWPB	Swap byte	SWPB, SWPAB, SWPALB, SWPLB
SWPH	Swap halfword	SWPH, SWPAH, SWPALH, SWPLH

C3.2.11.3 Compare and Swap operations

If FEAT_LSE2 is not implemented then:

- The CAS instructions require natural alignment.
- The CASP instructions require alignment to the total size of the memory being accessed.

If a compare and swap instruction does not perform a store, then the instruction does not have release semantics, regardless of the instruction ordering options.

For the CAS and CASP instructions, the architecture permits that a data read clears any Exclusives monitors associated with that location, even if the compare subsequently fails. If these instructions generate a synchronous Data Abort, the registers which are compared and loaded are restored to the values held in the registers before the instruction was executed.

If the destination register Rs is WZR or XZR, this is not regarded as doing a read for the purpose of a DMB LD barrier.

For more information on alignment requirements and behaviors, see Load-Exclusive/ Store-Exclusive and Atomic instructions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-38 Compare and swap instructions

Mnemonic	Instruction	See
CAS	Compare and swap	CAS, CASA, CASAL, CASL
CASB	Compare and swap byte	CASB, CASAB, CASALB, CASLB
CASH	Compare and swap halfword	CASH, CASAH, CASALH, CASLH
CASP	Compare and swap pair	CASP, CASPA, CASPAL, CASPL

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.3 Data processing - immediate

This section describes the instruction groups for data processing with immediate operands. It contains the following subsections:

- Arithmetic (immediate).
- Logical (immediate).
- Move (wide immediate).
- Move (immediate).
- PC-relative address calculation.
- Bitfield move.
- Bitfield insert and extract
- Extract register.
- Shift (immediate).
- Sign-extend and Zero-extend.

C3.3.1 Arithmetic (immediate)

The Arithmetic (immediate) instructions accept a 12-bit unsigned immediate value, optionally shifted left by 12 bits.

The Arithmetic (immediate) instructions that do not set Condition flags can read from and write to the current stack pointer. The flag setting instructions can read from the stack pointer, but they cannot write to it.

Table C3-39 shows the Arithmetic instructions with an immediate offset.

Table C3-39 Arithmetic instructions with an immediate

Mnemonic	Instruction	See
ADD	Add	ADD (immediate)
ADDS	Add and set flags	ADDS (immediate)
SUB	Subtract	SUB (immediate)
SUBS	Subtract and set flags	SUBS (immediate)
CMP	Compare	CMP (immediate)
CMN	Compare negative	CMN (immediate)

C3.3.2 Logical (immediate)

The Logical (immediate) instructions accept a bitmask immediate value that is a 32-bit pattern or a 64-bit pattern viewed as a vector of identical elements of size e = 2, 4, 8, 16, 32 or, 64 bits. Each element contains the same sub-pattern, that is a single run of 1 to (e - 1) nonzero bits from bit 0 followed by zero bits, then rotated by 0 to (e - 1) bits. This mechanism can generate 5334 unique 64-bit patterns as 2667 pairs of pattern and their bitwise inverse.

Note

Values that consist of only zeros or only ones cannot be described in this way.

The Logical (immediate) instructions that do not set the Condition flags can write to the current stack pointer, for example to align the stack pointer in a function prologue.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Apart from ANDS and its TST alias, Logical (immediate) instructions do not set the Condition flags. However, the final results of a bitwise operation can be tested by a CBZ, CBNZ, TBZ, or TBNZ conditional branch.

Table C3-40 shows the Logical immediate instructions.

Table C3-40 Logical immediate instructions

Mnemonic	Instruction	See
AND	Bitwise AND	AND (immediate)
ANDS	Bitwise AND and set flags	ANDS (immediate)
EOR	Bitwise exclusive OR	EOR (immediate)
ORR	Bitwise inclusive OR	ORR (immediate)
TST	Test bits	TST (immediate)

C3.3.3 Move (wide immediate)

The Move (wide immediate) instructions insert a 16-bit immediate, or inverted immediate, into a 16-bit aligned position in the destination register. The value of the other bits in the destination register depends on the variant used. The optional shift amount can be any multiple of 16 that is smaller than the register size.

Table C3-41 shows the Move (wide immediate) instructions.

Table C3-41 Move (wide immediate) instructions

Mnemonic	Instruction	See
MOVZ	Move wide with zero	MOVZ
MOVN	Move wide with NOT	MOVN
MOVK	Move wide with keep	MOVK

C3.3.4 Move (immediate)

The Move (immediate) instructions are aliases for a single MOVZ, MOVN, or ORR (immediate with zero register), instruction to load an immediate value into the destination register. An assembler must permit a signed or unsigned immediate, as long as its binary representation can be generated using one of these instructions, and an assembler error results if the immediate cannot be generated in this way. On disassembly, it is unspecified whether the immediate is output as a signed or an unsigned value.

If there is a choice between the MOVZ, MOVN, and ORR instruction to encode the immediate, then an assembler must prefer MOVZ to MOVN, and MOVZ or MOVN to ORR, to ensure reversability. A disassembler must output ORR (immediate with zero register) MOVZ, and MOVN, as a MOV mnemonic except that the underlying instruction must be used when:

- ORR has an immediate that can be generated by a MOVZ or MOVN instruction.
- A MOVN instruction has an immediate that can be encoded by MOVZ.
- MOVZ #0 or MOVN #0 have a shift amount other than LSL #0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-42 shows the Move (immediate) instructions.

Table C3-42 Move (immediate) instructions

Mnemonic	Instruction	See
MOV	Move (inverted wide immediate)	MOV (inverted wide immediate)
	Move (wide immediate)	MOV (wide immediate)
	Move (bitmask immediate)	MOV (bitmask immediate)

C3.3.5 PC-relative address calculation

The ADR instruction adds a signed, 21-bit immediate to the value of the program counter that fetched this instruction, and then writes the result to a general-purpose register. This permits the calculation of any byte address within ±1MB of the current PC.

The ADRP instruction shifts a signed, 21-bit immediate left by 12 bits, adds it to the value of the program counter with the bottom 12 bits cleared to zero, and then writes the result to a general-purpose register. This permits the calculation of the address at a 4KB aligned memory region. In conjunction with an ADD (immediate) instruction, or a load/store instruction with a 12-bit immediate offset, this allows for the calculation of, or access to, any address within ±4GB of the current PC.

Note

The term *page* used in the ADRP description is short-hand for the 4KB memory region, and is not related to the virtual memory translation granule size.

Table C3-43 shows the instructions used for PC-relative address calculations are as follows:

Table C3-43 PC-relative address calculation instructions

Mnemonic	Instruction	See
ADRP	Compute address of 4KB page at a PC-relative offset	ADRP
ADR	Compute address of label at a PC-relative offset.	ADR

C3.3.6 Bitfield move

The Bitfield move instructions copy a field of constant width from bit 0 in the source register to a constant bit position in the destination register, or from a constant bit position in the source register to bit 0 in the destination register. The remaining bits in the destination register are set as follows:

- For BFM, the remaining bits are unchanged.
- For UBFM the lower bits, if any, and upper bits, if any, are set to zero.
- For SBFM, the lower bits, if any, are set to zero, and the upper bits, if any, are set to a copy of the most-significant bit in the copied field.

Table C3-44 shows the Bitfield move instructions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-44 Bitfield move instructions

Mnemonic	Instruction	See
BFM	Bitfield move	BFM
SBFM	Signed bitfield move	SBFM
UBFM	Unsigned bitfield move (32-bit)	UBFM

C3.3.7 Bitfield insert and extract

The Bitfield insert and extract instructions are implemented as aliases of the Bitfield move instructions. Table C3-45 shows the Bitfield insert and extract aliases.

Table C3-45 Bitfield insert and extract instructions

Mnemonic	Instruction	See
BFC	Bitfield clear	BFC
BFI	Bitfield insert	BFI
BFXIL	Bitfield extract and insert low	BFXIL
SBFIZ	Signed bitfield insert in zero	SBFIZ
SBFX	Signed bitfield extract	SBFX
UBFIZ	Unsigned bitfield insert in zero	UBFIZ
UBFX	Unsigned bitfield extract	UBFX

C3.3.8 Extract register

Depending on the register width of the operands, the Extract register instruction copies a 32-bit or 64-bit field from a constant bit position within a double-width value formed by the concatenation of a pair of source registers to a destination register.

Table C3-46 shows the Extract (immediate) instructions.

Table C3-46 Extract register instructions

Mnemonic	Instruction	See
EXTR	Extract register from pair	EXTR

C3.3.9 Shift (immediate)

Shifts and rotates by a constant amount are implemented as aliases of the Bitfield move or Extract register instructions. The shift or rotate amount must be in the range 0 to one less than the register width of the instruction, inclusive.

Table C3-47 shows the aliases that can be used as immediate shift and rotate instructions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-47 Aliases for immediate shift and rotate instructions

Mnemonic	Instruction	See
ASR	Arithmetic shift right	ASR (immediate)
LSL	Logical shift left	LSL (immediate)
LSR	Logical shift right	LSR (immediate)
ROR	Rotate right	ROR (immediate)

C3.3.10 Sign-extend and Zero-extend

The Sign-extend and Zero-extend instructions are implemented as aliases of the Bitfield move instructions.

Table C3-48 shows the aliases that can be used as zero-extend and sign-extend instructions.

Table C3-48 Zero-extend and sign-extend instructions

Mnemonic	Instruction	See
SXTB	Sign-extend byte	SXTB
SXTH	Sign-extend halfword	SXTH
SXTW	Sign-extend word	SXTW
UXTB	Unsigned extend byte	UXTB
UXTH	Unsigned extend halfword	UXTH

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.4 Data processing - register

This section describes the instruction groups for data processing with all register operands. It contains the following subsections:

- Arithmetic (shifted register).
- Arithmetic (extended register).
- Arithmetic with carry.
- Flag manipulation instructions.
- Logical (shifted register).
- Move (register).
- Shift (register).
- Multiply and divide.
- CRC32.
- Bit operation.
- Conditional select.
- Conditional comparison.

C3.4.1 Arithmetic (shifted register)

The Arithmetic (shifted register) instructions apply an optional shift operator to the second source register value before performing the arithmetic operation. The register width of the instruction controls whether the new bits are fed into the intermediate result on a right shift or rotate at bit[63] or bit[31].

The shift operators `LSL`, `ASR`, and `LSR` accept an immediate shift amount in the range 0 to one less than the register width of the instruction, inclusive.

Omitting the shift operator implies `LSL #0`, which means that there is no shift. A disassembler must not output `LSL #0`. However, a disassembler must output all other shifts by zero.

The current stack pointer, SP or WSP, cannot be used with this class of instructions. See Arithmetic (extended register) for arithmetic instructions that can operate on the current stack pointer.

Table C3-49 shows the Arithmetic (shifted register) instructions.

Table C3-49 Arithmetic (shifted register) instructions

Mnemonic	Instruction	See
`ADD`	Add	ADD (shifted register)
`ADDS`	Add and set flags	ADDS (shifted register)
`SUB`	Subtract	SUB (shifted register)
`SUBS`	Subtract and set flags	SUBS (shifted register)
`CMN`	Compare negative	CMN (shifted register)
`CMP`	Compare	CMP (shifted register)
`NEG`	Negate	NEG (shifted register)
`NEGS`	Negate and set flags	NEGS

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.4.2 Arithmetic (extended register)

The extended register instructions provide an optional sign-extension or zero-extension of a portion of the second source register value, followed by an optional left shift by a constant amount of 1-4, inclusive.

The extended shift is described by the mandatory extend operator `SXTB`, `SXTH`, `SXTW`, `UXTB`, `UXTH`, or `UXTW` . This is followed by an optional left shift amount. If the shift amount is not specified, the default shift amount is zero. A disassembler must not output a shift amount of zero.

For 64-bit instruction forms, the additional operators `UXTX` and `SXTX` use all 64 bits of the second source register with an optional shift. In that case, Arm recommends `UXTX` as the operator. If and only if at least one register is SP, Arm recommends use of the `LSL` operator name, rather than `UXTX`, and when the shift amount is also zero then both the operator and the shift amount can be omitted. `UXTW` and `SXTW` both use all 32 bits of the second source register with an optional shift. In that case Arm recommends `UXTW` as the operator. If and only if at least one register is WSP, Arm recommends use of the `LSL` operator name, rather than `UXTW`, and when the shift amount is also zero then both the operator and the shift amount can be omitted.

For 32-bit instruction forms, the operators `UXTW` and `SXTW` both use all 32 bits of the second source register with an optional shift. In that case, Arm recommends `UXTW` as the operator. If and only if at least one register is WSP, Arm recommends use of the `LSL` operator name, rather than `UXTW`, and when the shift amount is also zero then both the operator and the shift amount can be omitted.

The non-flag setting variants of the extended register instruction permit the use of the current stack pointer as either the destination register and the first source register. The flag setting variants only permit the stack pointer to be used as the first source register.

In the 64-bit form of these instructions, the final register operand is written as `Wm` for all except the `UXTX`/`LSL` and `SXTX` extend operators. For example:

```
CMP X4, W5, SXTW
ADD X1, X2, W3, UXTB #2
SUB SP, SP, X1              // SUB SP, SP, X1, UXTX #0
```

Table C3-50 shows the Arithmetic (extended register) instructions.

Table C3-50 Arithmetic (extended register) instructions

Mnemonic	Instruction	See
`ADD`	Add	ADD (extended register)
`ADDS`	Add and set flags	ADDS (extended register)
`SUB`	Subtract	SUB (extended register)
`SUBS`	Subtract and set flags	SUBS (extended register)
`CMN`	Compare negative	CMN (extended register)
`CMP`	Compare	CMP (extended register)

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.4.3 Arithmetic with carry

The Arithmetic with carry instructions accept two source registers, with the carry flag as an additional input to the calculation. They do not support shifting of the second source register.

Table C3-51 shows the Arithmetic with carry instructions

Table C3-51 Arithmetic with carry instructions

Mnemonic	Instruction	See
ADC	Add with carry	ADC
ADCS	Add with carry and set flags	ADCS
SBC	Subtract with carry	SBC
SBCS	Subtract with carry and set flags	SBCS
NGC	Negate with carry	NGC
NGCS	Negate with carry and set flags	NGCS

C3.4.4 Flag manipulation instructions

The Flag manipulation instructions set the value of the NZCV condition flags directly.

The instructions SETF8 and SETF16 accept one source register and set the NZV condition flags based on the value of the input register. The instruction RMIF accepts one source register and two immediate values, rotating the first source register using the first immediate value and setting the NZCV condition flags masked by the second immediate value.

The instructions XAFLAG and AXFLAG convert PSTATE condition flags between the FCMP instruction format and an alternative format. See Table C6-1 for more information.

Table C3-52 shows the Flag manipulation instructions.

Table C3-52 Flag manipulation instructions

Mnemonic	Instruction	See
CFINV	Invert value of the PSTATE.C bit	CFINV
RMIF	Rotate, mask insert flags	RMIF
SETF8	Evaluation of 8-bit flags	SETF8, SETF16
SETF16	Evaluation of 16-bit flags	SETF8, SETF16

C3.4.5 Logical (shifted register)

The Logical (shifted register) instructions apply an optional shift operator to the second source register value before performing the main operation. The register width of the instruction controls whether the new bits are fed into the intermediate result on a right shift or rotate at bit[63] or bit[31].

The shift operators LSL, ASR, LSR, and ROR accept a constant immediate shift amount in the range 0 to one less than the register width of the instruction, inclusive.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Omitting the shift operator and amount implies `LSL #0`, which means that there is no shift. A disassembler must not output `LSL #0`. However, a disassembler must output all other shifts by zero.

Note

Apart from `ANDS`, `TST`, and `BICS`, the logical instructions do not set the Condition flags, but the final result of a bit operation can usually directly control a `CBZ`, `CBNZ`, `TBZ`, or `TBNZ` conditional branch.

Table C3-53 shows the Logical (shifted register) instructions.

Table C3-53 Logical (shifted register) instructions

Mnemonic	Instruction	See
`AND`	Bitwise AND	AND (shifted register)
`ANDS`	Bitwise AND and set flags	ANDS (shifted register)
`BIC`	Bitwise bit clear	BIC (shifted register)
`BICS`	Bitwise bit clear and set flags	BICS (shifted register)
`EON`	Bitwise exclusive-OR NOT	EON (shifted register)
`EOR`	Bitwise exclusive-OR	EOR (shifted register)
`ORR`	Bitwise inclusive OR	ORR (shifted register)
`MVN`	Bitwise NOT	MVN
`ORN`	Bitwise inclusive OR NOT	ORN (shifted register)
`TST`	Test bits	TST (shifted register)

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.4.6 Move (register)

The Move (register) instructions are aliases for other data processing instructions. They copy a value from a general-purpose register to another general-purpose register or the current stack pointer, or from the current stack pointer to a general-purpose register.

Table C3-54 MOV register instructions

Mnemonic	Instruction	See
MOV	Move register	MOV (register)
	Move register to SP or move SP to register	MOV (to/from SP)

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.4.7 Shift (register)

In the Shift (register) instructions, the shift amount is the positive value in the second source register modulo the register size. The register width of the instruction controls whether the new bits are fed into the result on a right shift or rotate at bit[63] or bit[31].

Table C3-55 shows the Shift (register) instructions.

Table C3-55 Shift (register) instructions

Mnemonic	Instruction	See
ASRV	Arithmetic shift right variable	ASRV
LSLV	Logical shift left variable	LSLV
LSRV	Logical shift right variable	LSRV
RORV	Rotate right variable	RORV

However, the Shift (register) instructions have a preferred set of aliases that match the shift immediate aliases described in Shift (immediate).

Table C3-56 shows the aliases for Shift (register) instructions.

Table C3-56 Aliases for Variable shift instructions

Mnemonic	Instruction	See
ASR	Arithmetic shift right	ASR (register)
LSL	Logical shift left	LSL (register)
LSR	Logical shift right	LSR (register)
ROR	Rotate right	ROR (register)

C3.4.8 Multiply and divide

This section describes the instructions used for integer multiplication and division. It contains the following subsections:

- Multiply.
- Divide.

C3.4.8.1 Multiply

The Multiply instructions write to a single 32-bit or 64-bit destination register, and are built around the fundamental four operand multiply-add and multiply-subtract operation, together with 32-bit to 64-bit widening variants. A 64-bit to 128-bit widening multiple can be constructed with two instructions, using SMULH or UMULH to generate the upper 64 bits. Table C3-57 shows the Multiply instructions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-57 Multiply integer instructions

Mnemonic	Instruction	See
MADD	Multiply-add	MADD
MSUB	Multiply-subtract	MSUB
MNEG	Multiply-negate	MNEG
MUL	Multiply	MUL
SMADDL	Signed multiply-add long	SMADDL
SMSUBL	Signed multiply-subtract long	SMSUBL
SMNEGL	Signed multiply-negate long	SMNEGL
SMULL	Signed multiply long	SMULL
SMULH	Signed multiply high	SMULH
UMADDL	Unsigned multiply-add long	UMADDL
UMSUBL	Unsigned multiply-subtract long	UMSUBL
UMNEGL	Unsigned multiply-negate long	UMNEGL
UMULL	Unsigned multiply long	UMULL
UMULH	Unsigned multiply high	UMULH

C3.4.8.2 Divide

The Divide instructions compute the quotient of a division, rounded towards zero. The remainder can then be computed as (numerator - (quotient × denominator)), using the MSUB instruction.

If a signed integer division (INT_MIN / -1) is performed where INT_MIN is the most negative integer value representable in the selected register size, then the result overflows the signed integer range. No indication of this overflow is produced and the result that is written to the destination register is INT_MIN.

A division by zero results in a zero being written to the destination register, without any indication that the division by zero occurred.

Table C3-58 shows the Divide instructions.

Table C3-58 Divide instructions

Mnemonic	Instruction	See
SDIV	Signed divide	SDIV
UDIV	Unsigned divide	UDIV

C3.4.9 CRC32

The CRC32 instructions operate on the general-purpose register file to update a 32-bit CRC value from an input value comprising 1, 2, 4, or 8 bytes. There are two different classes of CRC instructions, CRC32, and CRC32C, that support two commonly used 32-bit polynomials, known as CRC-32 and CRC-32C.

To fit with common usage, the bit order of the values is reversed as part of the operation.

When bits[19:16] of ID_AA64ISAR0_EL1 are set to 0b0001, the CRC instructions are implemented.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

These instructions are optional in an Armv8.0 implementation.

All implementations of Armv8.1 architecture and later are required to implement the `CRC32` instructions.

Table C3-59 shows the `CRC` instructions.

Table C3-59 CRC32 instructions

Mnemonic	Instruction	See
`CRC32B`	CRC-32 sum from byte	CRC32B, CRC32H, CRC32W, CRC32X
`CRC32H`	CRC-32 sum from halfword	CRC32B, CRC32H, CRC32W, CRC32X
`CRC32W`	CRC-32 sum from word	CRC32B, CRC32H, CRC32W, CRC32X
`CRC32X`	CRC-32 sum from doubleword	CRC32B, CRC32H, CRC32W, CRC32X
`CRC32CB`	CRC-32C sum from byte	CRC32CB, CRC32CH, CRC32CW, CRC32CX
`CRC32CH`	CRC-32C sum from halfword	CRC32CB, CRC32CH, CRC32CW, CRC32CX
`CRC32CW`	CRC-32C sum from word	CRC32CB, CRC32CH, CRC32CW, CRC32CX
`CRC32CX`	CRC-32C sum from doubleword	CRC32CB, CRC32CH, CRC32CW, CRC32CX

C3.4.10 Bit operation

Table C3-60 shows the Bit operation instructions.

Table C3-60 Bit operation instructions

Mnemonic	Instruction	See
`CLS`	Count leading sign bits	CLS
`CLZ`	Count leading zero bits	CLZ
`RBIT`	Reverse bit order	RBIT
`REV`	Reverse bytes in register	REV
`REV16`	Reverse bytes in halfwords	REV16
`REV32`	Reverse bytes in words	REV32
`REV64`	Reverse bytes in register	REV64

C3.4.11 Conditional select

The Conditional select instructions select between the first or second source register, depending on the current state of the Condition flags. When the named condition is true, the first source register is selected and its value is copied without modification to the destination register. When the condition is false the second source register is selected and its value might be optionally inverted, negated, or incremented by one, before writing to the destination register.

Other useful conditional set and conditional unary operations are implemented as aliases of the four Conditional select instructions.

Table C3-61 shows the Conditional select instructions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-61 Conditional select instructions

Mnemonic	Instruction	See
CSEL	Conditional select	CSEL
CSINC	Conditional select increment	CSINC
CSINV	Conditional select inversion	CSINV
CSNEG	Conditional select negation	CSNEG
CSET	Conditional set	CSET
CSETM	Conditional set mask	CSETM
CINC	Conditional increment	CINC
CINV	Conditional invert	CINV
CNEG	Conditional negate	CNEG

C3.4.12 Conditional comparison

The Conditional comparison instructions provide a conditional select for the NZCV Condition flags, setting the flags to the result of an arithmetic comparison of its two source register values if the named input condition is true, or to an immediate value if the input condition is false. There are register and immediate forms. The immediate form compares the source register to a small 5-bit unsigned value.

Table C3-62 shows the Conditional comparison instructions.

Table C3-62 Conditional comparison instructions

Mnemonic	Instruction	See
CCMN	Conditional compare negative (register)	CCMN (register)
CCMN	Conditional compare negative (immediate)	CCMN (immediate)
CCMP	Conditional compare (register)	CCMP (register)
CCMP	Conditional compare (immediate)	CCMP (immediate)

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.5 Data processing - SIMD and floating-point

This section describes the instruction groups for data processing with SIMD and floating-point register operands.

Common features of SIMD instructions gives general information about SIMD instructions.

The following subsections describe the scalar floating-point data processing instructions:

- Floating-point move (register).
- Floating-point move (immediate).
- Floating-point conversion.
- Floating-point round to integral value.
- Floating-point multiply-add.
- Floating-point arithmetic (one source).
- Floating-point arithmetic (two sources).
- Floating-point minimum and maximum.
- Floating-point comparison.
- Floating-point conditional select.

The following subsections describe the SIMD data processing instructions:

- SIMD move.
- SIMD arithmetic.
- SIMD compare.
- SIMD widening and narrowing arithmetic.
- SIMD unary arithmetic.
- SIMD by element arithmetic.
- SIMD permute.
- SIMD immediate.
- SIMD shift (immediate).
- SIMD conversions.
- SIMD reduce (across vector lanes).
- SIMD pairwise arithmetic.
- SIMD dot product.
- SIMD table lookup.
- SIMD complex number arithmetic instructions.
- SIMD multiply-accumulate.
- The Cryptographic Extension in AArch64 state.

For information about the floating-point exceptions, see Floating-point exceptions and exception traps.

C3.5.1 Common features of SIMD instructions

A number of SIMD instructions come in three forms:

Wide Indicated by the suffix *W*. The element width of the destination register and the first source operand is double that of the second source operand.

Long Indicated by the suffix *L*. The element width of the destination register is double that of both source operands.

Narrow Indicated by the suffix *N*. The element width of the destination register is half that of both source operands.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

In addition, each vector form of the instruction is part of a pair, with a second and upper part suffix of *2*, to identify the variant of the instruction:

- Where a SIMD operation widens or lengthens a 64-bit vector to a 128-bit vector, the instruction provides a second part operation that can extract the source from the upper 64 bits of the source registers.
- Where a SIMD operation narrows a 128-bit vector to a 64-bit vector, the instruction provides a second-part operation that can pack the result of a second operation into the upper part of the same destination register.

Note

This is referred to as a *lane set specifier*.

C3.5.2 Floating-point move (register)

The Floating-point move (register) instructions copy a scalar floating-point value from one register to another register without performing any conversion.

Some of the Floating-point move (register) instructions overlap with the functionality provided by the Advanced SIMD instructions `DUP`, `INS`, and `UMOV`. However, Arm recommends using the `FMOV` instructions when operating on scalar floating-point data to avoid the creation of scalar floating-point code that depends on the availability of the Advanced SIMD instruction set.

Table C3-63 shows the Floating-point move (register) instructions.

Table C3-63 Floating-point move (register) instructions

Mnemonic	Instruction	See
`FMOV`	Floating-point move register without conversion	FMOV (register)
	Floating-point move to or from general-purpose register without conversion	FMOV (general)

C3.5.3 Floating-point move (immediate)

The Floating-point move (immediate) instructions convert a small constant immediate floating-point value into a half-precision, single-precision, or double-precision scalar floating-point value in a SIMD and floating-point register.

The floating-point constant can be specified either in decimal notation, such as 12.0 or -1.2e1, or as a string beginning with `0x` followed by a hexadecimal representation of the IEEE 754 half-precision, single-precision, or double-precision encoding. Arm recommends that a disassembler uses the decimal notation, provided that this displays the value precisely.

Note

When FEAT_FP16 is not implemented, the only half-precision instructions that are supported are floating-point conversions between half-precision, single-precision, and double-precision.

The floating-point value must be expressible as $(\pm n/16 \times 2^r)$, where n is an integer in the range $16 \le n \le 31$ and r is an integer in the range of $-3 \le r \le 4$, that is a normalized binary floating-point encoding with one sign bit, four bits of fraction, and a 3-bit exponent.

Table C3-64 shows the Floating-point move (immediate) instruction:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-64 Floating-point move (immediate) instruction

Mnemonic	Instruction	See
FMOV	Floating-point move immediate	FMOV (scalar, immediate)

C3.5.4 Floating-point conversion

The following subsections describe the conversion of floating-point values:

- Convert floating-point precision.
- Convert from floating-point single-precision to BFloat16.
- Convert from floating-point to integer or fixed-point.
- Convert from integer or fixed-point to floating-point.

C3.5.4.1 Convert floating-point precision

These instructions convert a floating-point scalar with one precision to a floating-point scalar with a different precision, using the current rounding mode as specified by FPCR.RMode.

Table C3-65 shows the Floating-point precision conversion instruction.

Table C3-65 Floating-point precision conversion instruction

Mnemonic	Instruction	See
FCVT	Floating-point convert precision (scalar)	FCVT

C3.5.4.2 Convert from floating-point single-precision to BFloat16

The BFCVT instruction is provided by FEAT_BF16. This instruction converts a single-precision floating-point input to BFloat16 format, honoring the FPCR rounding mode controls to give a more accurate conversion than simply removing the bottom 16 bits of the input.

Table C3-66 shows this instruction.

Table C3-66 Floating-point single-precision to BFloat16 conversion instruction

Mnemonic	Instruction	See
BFCVT	BFloat16 floating-point convert from single-precision to BFloat16 format (scalar)	BFCVT

C3.5.4.3 Convert from floating-point to integer or fixed-point

Floating-point to integer or fixed-point conversion is supported by the following instruction variants:

- Scalar instruction variant, which converts a floating-point value in a SIMD and floating-point source register to an integer or fixed-point value in a general-purpose destination register.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Scalar SIMD&FP instruction variant, which converts a floating-point value in a SIMD and floating-point source register to an integer value in a SIMD and floating-point destination register.

For a fixed-point value, a final immediate operand indicates that the general-purpose register holds a fixed-point number and `fbits` indicates the number of bits after the binary point. `fbits` is in the range 1- 32 inclusive for a 32-bit general-purpose register name, and 1-64 inclusive for a 64-bit general-purpose register name.

Table C3-67 shows the floating-point to integer or fixed-point conversion instructions.

Table C3-67 Floating-point to integer or fixed-point conversion instructions

Mnemonic	Instruction	See
FCVTAS	Floating-point scalar convert to signed integer, rounding to nearest with ties to away (scalar form)	FCVTAS (scalar)
FCVTAU	Floating-point scalar convert to unsigned integer, rounding to nearest with ties to away (scalar form)	FCVTAU (scalar)
FCVTMS	Floating-point scalar convert to signed integer, rounding toward minus infinity (scalar form)	FCVTMS (scalar)
FCVTMU	Floating-point scalar convert to unsigned integer, rounding toward minus infinity (scalar form)	FCVTMU (scalar)
FCVTNS	Floating-point scalar convert to signed integer, rounding to nearest with ties to even (scalar form)	FCVTNS (scalar)
FCVTNU	Floating-point scalar convert to unsigned integer, rounding to nearest with ties to even (scalar form)	FCVTNU (scalar)
FCVTPS	Floating-point scalar convert to signed integer, rounding toward positive infinity (scalar form)	FCVTPS (scalar)
FCVTPU	Floating-point scalar convert to unsigned integer, rounding toward positive infinity (scalar form)	FCVTPU (scalar)
FCVTZS	Floating-point scalar convert to signed integer, rounding toward zero (scalar form)	FCVTZS (scalar, integer)
	Floating-point scalar convert to signed fixed-point, rounding toward zero (scalar form)	FCVTZS (scalar, fixed-point)
FCVTZU	Floating-point scalar convert to unsigned integer, rounding toward zero (scalar form)	FCVTZU (scalar, integer)
	Floating-point scalar convert to unsigned fixed-point, rounding toward zero (scalar form)	FCVTZU (scalar, fixed-point)
FJCVTZS	Floating-point Javascript convert to signed fixed-point, rounding toward zero (scalar form)	FJCVTZS

C3.5.4.4 Convert from integer or fixed-point to floating-point

The integer or fixed-point to floating-point conversion is supported by the following instruction variants:

- Scalar instruction variant, which converts an integer or fixed-point value in a general-purpose register to a floating-point value in a SIMD and floating-point destination register.
- Scalar SIMD&FP variant, which converts an integer value in a SIMD and floating-point source register to a floating-point value in a SIMD and floating-point destination register.

For a fixed-point value, a final immediate operand indicates that the general-purpose register holds a fixed-point number and `fbits` indicates the number of bits after the binary point. `fbits` is in the range 1- 32 inclusive for a 32-bit general-purpose register name, and 1-64 inclusive for a 64-bit general-purpose register name.

Table C3-68 shows the integer or fixed-point to floating-point conversion instructions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-68 Integer or fixed-point to floating-point conversion instructions

Mnemonic	Instruction	See
SCVTF	Signed integer scalar convert to floating-point, using the current rounding mode (scalar form)	SCVTF (scalar, integer)
	Signed fixed-point convert to floating-point, using the current rounding mode (scalar form)	SCVTF (scalar, fixed-point)
UCVTF	Unsigned integer scalar convert to floating-point, using the current rounding mode (scalar form)	UCVTF (scalar, integer)
	Unsigned fixed-point convert to floating-point, using the current rounding mode (scalar form)	UCVTF (scalar, fixed-point)

C3.5.5 Floating-point round to integral value

The following subsections describe instructions which round a floating-point number to an integral valued floating-point number in the same format:

- Floating-point round to an integer of the same size as the register
- Floating-point round to 32-bit or 64-bit integer

C3.5.5.1 Floating-point round to an integer of the same size as the register

The following instructions round a floating-point value to an integer floating-point value of the same size.

For these instructions:

- A zero input gives a zero result with the same sign.
- An infinite input gives an infinite result with the same sign.
- A NaN is propagated as in normal floating-point arithmetic.

These instructions can cause floating-point exceptions, for more information see Floating-point exceptions and exception traps.

Table C3-69 shows the Floating-point round to integer instructions.

Table C3-69 Floating-point round to integer instructions

Mnemonic	Instruction	See
FRINTA	Floating-point round to integer, to nearest with ties to away	FRINTA (scalar)
FRINTI	Floating-point round to integer, using current rounding mode	FRINTI (scalar)
FRINTM	Floating-point round to integer, toward minus infinity	FRINTM (scalar)
FRINTN	Floating-point round to integer, to nearest with ties to even	FRINTN (scalar)
FRINTP	Floating-point round to integer, toward positive infinity	FRINTP (scalar)
FRINTX	Floating-point round to integer exact, using current rounding mode	FRINTX (scalar)
FRINTZ	Floating-point round to integer, toward zero	FRINTZ (scalar)

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.5.6 Floating-point multiply-add

Table C3-70 shows the Floating-point multiply-add instructions that require three source register operands.

Table C3-70 Floating-point multiply-add instructions

Mnemonic	Instruction	See
FMADD	Floating-point scalar fused multiply-add	FMADD
FMSUB	Floating-point scalar fused multiply-subtract	FMSUB
FNMADD	Floating-point scalar negated fused multiply-add	FNMADD
FNMSUB	Floating-point scalar negated fused multiply-subtract	FNMSUB

C3.5.7 Floating-point arithmetic (one source)

Table C3-71 shows the Floating-point arithmetic instructions that require a single source register operand.

Table C3-71 Floating-point arithmetic instructions with one source register

Mnemonic	Instructions	See
FABS	Floating-point scalar absolute value	FABS (scalar)
FNEG	Floating-point scalar negate	FNEG (scalar)
FSQRT	Floating-point scalar square root	FSQRT (scalar)

C3.5.8 Floating-point arithmetic (two sources)

Table C3-72 shows the Floating-point arithmetic instructions that require two source register operands.

Table C3-72 Floating-point arithmetic instructions with two source registers

Mnemonic	Instruction	See
FADD	Floating-point scalar add	FADD (scalar)
FDIV	Floating-point scalar divide	FDIV (scalar)
FMUL	Floating-point scalar multiply	FMUL (scalar)
FNMUL	Floating-point scalar multiply-negate	FNMUL (scalar)
FSUB	Floating-point scalar subtract	FSUB (scalar)

C3.5.9 Floating-point minimum and maximum

The `min(x,y)` and `max(x,y)` operations return a quiet NaN when either x or y is NaN.

As described in Flushing denormalized numbers to zero, if flushing denormalized inputs to zero is enabled, denormal operands are flushed to zero before comparison, and if the result of the comparison is the flushed value, then a zero value

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

is returned. Where both *x* and *y* are zero, or denormal values flushed to zero, with different signs, then +0.0 is returned by `max()` and -0.0 by `min()`.

The `minNum(x,y)` and `maxNum(x,y)` operations follow the IEEE 754-2008 standard and return the numerical operand when one operand is numerical and the other a quiet NaN. Apart from this additional handling of a single quiet NaN, the result is then identical to `min(x,y)` and `max(x,y)`.

Table C3-73 shows the Floating-point instructions that can perform floating-point minimum and maximum operations.

Table C3-73 Floating-point minimum and maximum instructions

Mnemonic	Instruction	See
FMAX	Floating-point scalar maximum	FMAX (scalar)
FMAXNM	Floating-point scalar maximum number	FMAXNM (scalar)
FMIN	Floating-point scalar minimum	FMIN (scalar)
FMINNM	Floating-point scalar minimum number	FMINNM (scalar)

C3.5.10 Floating-point comparison

These instructions set the NZCV Condition flags in PSTATE, based on the result of a comparison of two operands. If the floating-point comparisons are *unordered*, where one or both operands are a form of NaN, the C and V bits are set to 1 and the N and Z bits are cleared to 0.

Note

The A64 floating-point comparison instructions do not change the Condition flags in the FPSR.

For the conditional Floating-point comparison instructions, if the condition is TRUE, the flags are updated to the result of the comparison, otherwise the flags are updated to the immediate value that is defined in the instruction encoding.

The quiet compare instructions generate an Invalid Operation floating-point exception if either of the source operands is a signaling NaN. The signaling compare instructions generate an Invalid Operation floating-point exception if either of the source operands is any type of NaN.

Table C3-74 shows the Floating-point comparison instructions.

Table C3-74 Floating-point comparison instructions

Mnemonic	Instruction	See
FCMP	Floating-point quiet compare	FCMP
FCMPE	Floating-point signaling compare	FCMPE
FCCMP	Floating-point conditional quiet compare	FCCMP
FCCMPE	Floating-point conditional signaling compare	FCCMPE

C3.5.11 Floating-point conditional select

Table C3-75 shows the Floating-point conditional select instructions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-75 Floating-point conditional select instructions

Mnemonic	Instruction	See
FCSEL	Floating-point scalar conditional select	FCSEL

C3.5.12 SIMD move

The functionality of some data movement instructions overlaps with that provided by the scalar floating-point FMOV instructions described in Floating-point move (register).

Table C3-76 shows the SIMD move instructions.

Table C3-76 SIMD move instructions

Mnemonic	Instruction	See
DUP	Duplicate vector element to vector or scalar	DUP (element)
DUP	Duplicate general-purpose register to vector	DUP (general)
INS[a]	Insert vector element from another vector element	INS (element)
	Insert vector element from general-purpose register	INS (general)
MOV	Move vector element to vector element	MOV (element)
	Move general-purpose register to vector element	MOV (from general)
	Move vector element to scalar	MOV (scalar)
	Move vector element to general-purpose register	MOV (to general)
UMOV	Unsigned move vector element to general-purpose register	UMOV
SMOV	Signed move vector element to general-purpose register	SMOV

a. Disassembles as MOV.

C3.5.13 SIMD arithmetic

Table C3-77 shows the SIMD arithmetic instructions.

Table C3-77 SIMD arithmetic instructions

Mnemonic	Instruction	See
ADD	Add (vector and scalar form)	ADD (vector)
AND	Bitwise AND (vector form)	AND (vector)
BIC	Bitwise bit clear (register) (vector form)	BIC (vector, register)
BIF	Bitwise insert if false (vector form)	BIF
BIT	Bitwise insert if true (vector form)	BIT
BSL	Bitwise select (vector form)	BSL
EOR	Bitwise exclusive-OR (vector form)	EOR (vector)

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
FABD	Floating-point absolute difference (vector and scalar form)	FABD
FADD	Floating-point add (vector form)	FADD (vector)
FDIV	Floating-point divide (vector form)	FDIV (vector)
FMAX	Floating-point maximum (vector form)	FMAX (vector)
FMAXNM	Floating-point maximum number (vector form)	FMAXNM (vector)
FMIN	Floating-point minimum (vector form)	FMIN (vector)
FMINNM	Floating-point minimum number (vector form)	FMINNM (vector)
FMUL	Floating-point multiply (vector form)	FMUL (vector)
FMULX	Floating-point multiply extended (vector and scalar form)	FMULX
FRECPS	Floating-point reciprocal step (vector and scalar form)	FRECPS
FRSQRTS	Floating-point reciprocal square root step (vector and scalar form)	FRSQRTS
FSUB	Floating-point subtract (vector form)	FSUB (vector)
MLA	Multiply-add (vector form)	MLA (vector)
MLS	Multiply-subtract (vector form)	MLS (vector)
MUL	Multiply (vector form)	MUL (vector)
MOV	Move vector register (vector form)	MOV (vector)
ORN	Bitwise inclusive OR NOT (vector form)	ORN (vector)
ORR	Bitwise inclusive OR (register) (vector form)	ORR (vector, register)
PMUL	Polynomial multiply (vector form)	PMUL
SABA	Signed absolute difference and accumulate (vector form)	SABA
SABD	Signed absolute difference (vector form)	SABD
SHADD	Signed halving add (vector form)	SHADD
SHSUB	Signed halving subtract (vector form)	SHSUB
SMAX	Signed maximum (vector form)	SMAX
SMIN	Signed minimum (vector form)	SMIN
SQADD	Signed saturating add (vector and scalar form)	SQADD
SQDMULH	Signed saturating doubling multiply returning high half (vector and scalar form)	SQDMULH (vector)
SQRSHL	Signed saturating rounding shift left (register) (vector and scalar form)	SQRSHL
SQRDMLAH	Signed saturating rounding doubling multiply accumulate returning high half	SQRDMLAH (vector)
SQRDMLSH	Signed saturating rounding doubling multiply subtract returning high half	SQRDMLSH (vector)
SQRDMULH	Signed saturating rounding doubling multiply returning high half (vector and scalar form)	SQRDMULH (vector)
SQSHL	Signed saturating shift left (register) (vector and scalar form)	SQSHL (register)
SQSUB	Signed saturating subtract (vector and scalar form)	SQSUB
SRHADD	Signed rounding halving add (vector form)	SRHADD
SRSHL	Signed rounding shift left (register) (vector and scalar form)	SRSHL
SSHL	Signed shift left (register) (vector and scalar form)	SSHL
SUB	Subtract (vector and scalar form)	SUB (vector)
UABA	Unsigned absolute difference and accumulate (vector form)	UABA
UABD	Unsigned absolute difference (vector form)	UABD

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
UHADD	Unsigned halving add (vector form)	UHADD
UHSUB	Unsigned halving subtract (vector form)	UHSUB
UMAX	Unsigned maximum (vector form)	UMAX
UMIN	Unsigned minimum (vector form)	UMIN
UQADD	Unsigned saturating add (vector and scalar form)	UQADD
UQRSHL	Unsigned saturating rounding shift left (register) (vector and scalar form)	UQRSHL
UQSHL	Unsigned saturating shift left (register) (vector and scalar form)	UQSHL (register)
UQSUB	Unsigned saturating subtract (vector and scalar form)	UQSUB
URHADD	Unsigned rounding halving add (vector form)	URHADD
URSHL	Unsigned rounding shift left (register) (vector and scalar form)	URSHL
USHL	Unsigned shift left (register) (vector and scalar form)	USHL

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.5.14 SIMD compare

The SIMD compare instructions compare vector or scalar elements according to the specified condition and set the destination vector element to all ones if the condition holds, or to zero if the condition does not hold.

Note

Some of the comparisons, such as LS, LE, LO, and LT, can be made by reversing the operands and using the opposite comparison, HS, GE, HI, or GT.

Table C3-78 shows that SIMD compare instructions.

Table C3-78 SIMD compare instructions

Mnemonic	Instruction	See
CMEQ	Compare bitwise equal (vector and scalar form)	CMEQ (register)
	Compare bitwise equal to zero (vector and scalar form)	CMEQ (zero)
CMHS	Compare unsigned higher or same (vector and scalar form)	CMHS (register)
CMGE	Compare signed greater than or equal (vector and scalar form)	CMGE (register)
	Compare signed greater than or equal to zero (vector and scalar form)	CMGE (zero)
CMHI	Compare unsigned higher (vector and scalar form)	CMHI (register)
CMGT	Compare signed greater than (vector and scalar form)	CMGT (register)
	Compare signed greater than zero (vector and scalar form)	CMGT (zero)
CMLE	Compare signed less than or equal to zero (vector and scalar form)	CMLE (zero)
CMLT	Compare signed less than zero (vector and scalar form)	CMLT (zero)
CMTST	Compare bitwise test bits nonzero (vector and scalar form)	CMTST
FCMEQ	Floating-point compare equal (vector and scalar form)	FCMEQ (register)
	Floating-point compare equal to zero (vector and scalar form)	FCMEQ (zero)
FCMGE	Floating-point compare greater than or equal (vector and scalar form)	FCMGE (register)
	Floating-point compare greater than or equal to zero (vector and scalar form)	FCMGE (zero)
FCMGT	Floating-point compare greater than (vector and scalar form)	FCMGT (register)
	Floating-point compare greater than zero (vector and scalar form)	FCMGT (zero)
FCMLE	Floating-point compare less than or equal to zero (vector and scalar form)	FCMLE (zero)
FCMLT	Floating-point compare less than zero (vector and scalar form)	FCMLT (zero)
FACGE	Floating-point absolute compare greater than or equal (vector and scalar form)	FACGE
FACGT	Floating-point absolute compare greater than (vector and scalar form)	FACGT

C3.5.15 SIMD widening and narrowing arithmetic

For information about the variants of these instructions, see Common features of SIMD instructions.

Table C3-79 shows the SIMD widening and narrowing arithmetic instructions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-79 SIMD widening and narrowing arithmetic instructions

Mnemonic	Instruction	See
ADDHN, ADDHN2	Add returning high, narrow (vector form)	ADDHN, ADDHN2
PMULL, PMULL2	Polynomial multiply long (vector form)	PMULL, PMULL2
RADDHN, RADDHN2	Rounding add returning high, narrow (vector form)	RADDHN, RADDHN2
RSUBHN, RSUBHN2	Rounding subtract returning high, narrow (vector form)	RSUBHN, RSUBHN2
SABAL, SABAL2	Signed absolute difference and accumulate long (vector form)	SABAL, SABAL2
SABDL, SABDL2	Signed absolute difference long (vector form)	SABDL, SABDL2
SADDL, SADDL2	Signed add long (vector form)	SADDL, SADDL2
SADDW, SADDW2	Signed add wide (vector form)	SADDW, SADDW2
SMLAL, SMLAL2	Signed multiply-add long (vector form)	SMLAL, SMLAL2 (vector)
SMLSL, SMLSL2	Signed multiply-subtract long (vector form)	SMLSL, SMLSL2 (vector)
SMULL, SMULL2	Signed multiply long (vector form)	SMULL, SMULL2 (vector)
SQDMLAL, SQDMLAL2	Signed saturating doubling multiply-add long (vector and scalar form)	SQDMLAL, SQDMLAL2 (vector)
SQDMLSL, SQDMLSL2	Signed saturating doubling multiply-subtract long (vector and scalar form)	SQDMLSL, SQDMLSL2 (vector)
SQDMULL, SQDMULL2	Signed saturating doubling multiply long (vector and scalar form)	SQDMULL, SQDMULL2 (vector)
SSUBL, SSUBL2	Signed subtract long (vector form)	SSUBL, SSUBL2
SSUBW, SSUBW2	Signed subtract wide (vector form)	SSUBW, SSUBW2
SUBHN, SUBHN2	Subtract returning high, narrow (vector form)	SUBHN, SUBHN2
UABAL, UABAL2	Unsigned absolute difference and accumulate long (vector form)	UABAL, UABAL2
UABDL, UABDL2	Unsigned absolute difference long (vector form)	UABDL, UABDL2
UADDL, UADDL2	Unsigned add long (vector form)	UADDL, UADDL2
UADDW, UADDW2	Unsigned add wide (vector form)	UADDW, UADDW2
UMLAL, UMLAL2	Unsigned multiply-add long (vector form)	UMLAL, UMLAL2 (vector)
UMLSL, UMLSL2	Unsigned multiply-subtract long (vector form)	UMLSL, UMLSL2 (vector)
UMULL, UMULL2	Unsigned multiply long (vector form)	UMULL, UMULL2 (vector)
USUBL, USUBL2	Unsigned subtract long (vector form)	USUBL, USUBL2
USUBW, USUBW2	Unsigned subtract wide (vector form)	USUBW, USUBW2

C3.5.16 SIMD unary arithmetic

For information about the variants of these instructions, see Common features of SIMD instructions.

Table C3-80 shows the SIMD unary arithmetic instructions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-80 SIMD unary arithmetic instructions

Mnemonic	Instruction	See
ABS	Absolute value (vector and scalar form)	ABS
CLS	Count leading sign bits (vector form)	CLS (vector)
CLZ	Count leading zero bits (vector form)	CLZ (vector)
CNT	Population count per byte (vector form)	CNT
FABS	Floating-point absolute (vector form)	FABS (vector)
FNEG	Floating-point negate (vector form)	FNEG (vector)
FRECPE	Floating-point reciprocal estimate (vector and scalar form)	FRECPE
FRECPX	Floating-point reciprocal exponent (scalar form)	FRECPX
FRINTA	Floating-point round to integer, to nearest with ties to away (vector form)	FRINTA (vector)
FRINTI	Floating-point round to integer, using current rounding mode (vector form)	FRINTI (vector)
FRINTM	Floating-point round to integer, toward minus infinity (vector form)	FRINTM (vector)
FRINTN	Floating-point round to integer, to nearest with ties to even (vector form)	FRINTN (vector)
FRINTP	Floating-point round to integer, toward positive infinity (vector form)	FRINTP (vector)
FRINTX	Floating-point round to integer exact, using current rounding mode (vector form)	FRINTX (vector)
FRINTZ	Floating-point round to integer, toward zero (vector form)	FRINTZ (vector)
FRSQRTE	Floating-point reciprocal square root estimate (vector and scalar form)	FRSQRTE
FSQRT	Floating-point square root (vector form)	FSQRT (vector)
MVN	Bitwise NOT (vector form)	MVN
NEG	Negate (vector and scalar form)	NEG (vector)
NOT	Bitwise NOT (vector form)	NOT
RBIT	Bitwise reverse (vector form)	RBIT (vector)
REV16	Reverse elements in 16-bit halfwords (vector form)	REV16 (vector)
REV32	Reverse elements in 32-bit words (vector form)	REV32 (vector)
REV64	Reverse elements in 64-bit doublewords (vector form)	REV64
SADALP	Signed add and accumulate long pairwise (vector form)	SADALP
SADDLP	Signed add long pairwise (vector form)	SADDLP
SQABS	Signed saturating absolute value (vector and scalar form)	SQABS
SQNEG	Signed saturating negate (vector and scalar form)	SQNEG
SQXTN, SQXTN2	Signed saturating extract narrow (vector form)	SQXTN, SQXTN2
SQXTUN, SQXTUN2	Signed saturating extract unsigned narrow (vector and scalar form)	SQXTUN, SQXTUN2
SUQADD	Signed saturating accumulate of unsigned value (vector and scalar form)	SUQADD
SXTL, SXTL2	Signed extend long	SXTL, SXTL2
UADALP	Unsigned add and accumulate long pairwise (vector form)	UADALP
UADDLP	Unsigned add long pairwise (vector form)	UADDLP
UQXTN, UQXTN2	Unsigned saturating extract narrow (vector form)	UQXTN, UQXTN2
URECPE	Unsigned reciprocal estimate (vector form)	URECPE

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
URSQRTE	Unsigned reciprocal square root estimate (vector form)	URSQRTE
USQADD	Unsigned saturating accumulate of signed value (vector and scalar form)	USQADD
UXTL, UXTL2	Unsigned extend long	UXTL, UXTL2
XTN, XTN2	Extract narrow (vector form)	XTN, XTN2

C3.5.17 SIMD by element arithmetic

For information about the variants of these instructions, see Common features of SIMD instructions.

Table C3-81 shows the SIMD by element arithmetic instructions.

Table C3-81 SIMD by element arithmetic instructions

Mnemonic	Instruction	See
FMLA	Floating-point fused multiply-add (vector and scalar form)	FMLA (by element)
FMLAL, FMLAL2	Floating-point fused multiply-add long (vector form)	FMLAL, FMLAL2 (by element)
FMLS	Floating-point fused multiply-subtract (vector and scalar form)	FMLS (by element).
FMLSL, FMLSL2	Floating-point fused multiply-subtract long (vector form)	FMLSL, FMLSL2 (by element)
FMUL	Floating-point multiply (vector and scalar form)	FMUL (by element)
FMULX	Floating-point multiply extended (vector and scalar form)	FMULX (by element)
MLA	Multiply-add (vector form)	MLA (by element)
MLS	Multiply-subtract (vector form)	MLS (by element)
MUL	Multiply (vector form)	MUL (by element)
SMLAL, SMLAL2	Signed multiply-add long (vector form)	SMLAL, SMLAL2 (by element)
SMLSL, SMLSL2	Signed multiply-subtract long (vector form)	SMLSL, SMLSL2 (by element)
SMULL, SMULL2	Signed multiply long (vector form)	SMULL, SMULL2 (by element)
SQDMLAL, SQDMLAL2	Signed saturating doubling multiply-add long (vector and scalar form)	SQDMLAL, SQDMLAL2 (by element)
SQDMLSL, SQDMLSL2	Signed saturating doubling multiply-subtract long (vector form)	SQDMLSL, SQDMLSL2 (by element)
SQDMULH	Signed saturating doubling multiply returning high half (vector and scalar form)	SQDMULH (by element)
SQDMULL, SQDMULL2	Signed saturating doubling multiply long (vector and scalar form)	SQDMULL, SQDMULL2 (by element)
SQRDMLAH	Signed saturating rounding doubling multiply accumulate returning high half	SQRDMLSH (by element)
SQRDMLSH	Signed saturating rounding doubling multiply subtract returning high half	SQRDMLSH (vector)
SQRDMULH	Signed saturating rounding doubling multiply returning high half (vector and scalar form)	SQRDMULH (by element)
UMLAL, UMLAL2	Unsigned multiply-add long (vector form)	UMLAL, UMLAL2 (by element)
UMLSL, UMLSL2	Unsigned multiply-subtract long (vector form)	UMLSL, UMLSL2 (by element)
UMULL, UMULL2	Unsigned multiply long (vector form)	UMULL, UMULL2 (by element)

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.5.18 SIMD permute

Table C3-82 shows the SIMD permute instructions.

Table C3-82 SIMD permute instructions

Mnemonic	Instruction	See
EXT	Extract vector from a pair of vectors	EXT
TRN1	Transpose vectors (primary)	TRN1
TRN2	Transpose vectors (secondary)	TRN2
UZP1	Unzip vectors (primary)	UZP1
UZP2	Unzip vectors (secondary)	UZP2
ZIP1	Zip vectors (primary)	ZIP1
ZIP2	Zip vectors (secondary)	ZIP2

C3.5.19 SIMD immediate

Table C3-83 shows the SIMD immediate instructions.

Table C3-83 SIMD immediate instructions

Mnemonic	Instruction	See
BIC	Bitwise bit clear immediate	BIC (vector, immediate)
FMOV	Floating-point move immediate	FMOV (vector, immediate)
MOVI	Move immediate	MOVI
MVNI	Move inverted immediate	MVNI
ORR	Bitwise inclusive OR immediate	ORR (vector, immediate)

C3.5.20 SIMD shift (immediate)

For information about the variants of these instructions, see Common features of SIMD instructions.

Table C3-84 shows the SIMD shift immediate instructions.

Table C3-84 SIMD shift (immediate) instructions

Mnemonic	Instruction	See
RSHRN, RSHRN2	Rounding shift right narrow immediate (vector form)	RSHRN, RSHRN2
SHL	Shift left immediate (vector and scalar form)	SHL

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
SHLL, SHLL2	Shift left long (by element size) (vector form)	SHLL, SHLL2
SHRN, SHRN2	Shift right narrow immediate (vector form)	SHRN, SHRN2
SLI	Shift left and insert immediate (vector and scalar form)	SLI
SQRSHRN, SQRSHRN2	Signed saturating rounded shift right narrow immediate (vector and scalar form)	SQRSHRN, SQRSHRN2
SQRSHRUN, SQRSHRUN2	Signed saturating shift right unsigned narrow immediate (vector and scalar form)	SQRSHRUN, SQRSHRUN2
SQSHL	Signed saturating shift left immediate (vector and scalar form)	SQSHL (immediate)
SQSHLU	Signed saturating shift left unsigned immediate (vector and scalar form)	SQSHLU
SQSHRN, SQSHRN2	Signed saturating shift right narrow immediate (vector and scalar form)	SQSHRN, SQSHRN2
SQSHRUN, SQSHRUN2	Signed saturating shift right unsigned narrow immediate (vector and scalar form)	SQSHRUN, SQSHRUN2
SRI	Shift right and insert immediate (vector and scalar form)	SRI
SRSHR	Signed rounding shift right immediate (vector and scalar form)	SRSHR
SRSRA	Signed rounding shift right and accumulate immediate (vector and scalar form)	SRSRA.
SSHLL, SSHLL2	Signed shift left long immediate (vector form)	SSHLL, SSHLL2
SSHR	Signed shift right immediate (vector and scalar form)	SSHR
SSRA	Signed integer shift right and accumulate immediate (vector and scalar form)	SSRA
SXTL, SXTL2	Signed integer extend (vector only)	SXTL, SXTL2
UQRSHRN, UQRSHRN2	Unsigned saturating rounded shift right narrow immediate (vector and scalar form)	UQRSHRN, UQRSHRN2
UQSHL	Unsigned saturating shift left immediate (vector and scalar form)	UQSHL (immediate)
UQSHRN, UQSHRN2	Unsigned saturating shift right narrow immediate (vector and scalar form)	UQSHRN, UQSHRN2
URSHR	Unsigned rounding shift right immediate (vector and scalar form)	URSHR
URSRA	Unsigned integer rounding shift right and accumulate immediate (vector and scalar form)	URSRA
USHLL, USHLL2	Unsigned shift left long immediate (vector form)	USHLL, USHLL2
USHR	Unsigned shift right immediate (vector and scalar form)	USHR
USRA	Unsigned shift right and accumulate immediate (vector and scalar form)	USRA
UXTL, UXTL2	Unsigned integer extend (vector only)	UXTL, UXTL2

C3.5.21 SIMD conversions

C3.5.21.1 Floating-point conversion

Table C3-85 shows the SIMD floating-point conversion instructions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-85 SIMD floating-point conversion instructions

Mnemonic	Instruction	See
FCVTAS	Floating-point convert to signed integer, rounding to nearest with ties to away (vector and scalar form)	FCVTAS (vector)
FCVTAU	Floating-point convert to unsigned integer, rounding to nearest with ties to away (vector and scalar form)	FCVTAU (vector)
FCVTL, FCVTL2	Floating-point convert to higher precision long (vector form)	FCVTL, FCVTL2
FCVTMS	Floating-point convert to signed integer, rounding toward minus infinity (vector and scalar form)	FCVTMS (vector)
FCVTMU	Floating-point convert to unsigned integer, rounding toward minus infinity (vector and scalar form)	FCVTMU (vector)
FCVTN, FCVTN2	Floating-point convert to lower precision narrow (vector form)	FCVTN, FCVTN2 (double to single-precision, single to half-precision)
FCVTNS	Floating-point convert to signed integer, rounding to nearest with ties to even (vector and scalar form)	FCVTNS (vector)
FCVTNU	Floating-point convert to unsigned integer, rounding to nearest with ties to even (vector and scalar form)	FCVTNU (vector)
FCVTPS	Floating-point convert to signed integer, rounding toward positive infinity (vector and scalar form)	FCVTPS (vector)
FCVTPU	Floating-point convert to unsigned integer, rounding toward positive infinity (vector and scalar form)	FCVTPU (vector)
FCVTXN, FCVTXN2	Floating-point convert to lower precision narrow, rounding to odd (vector and scalar form)	FCVTXN, FCVTXN2
FCVTZS	Floating-point convert to signed integer, rounding toward zero (vector and scalar form)	FCVTZS (vector, integer)
	Floating-point convert to signed fixed-point, rounding toward zero (vector and scalar form)	FCVTZS (vector, fixed-point)
FCVTZU	Floating-point convert to unsigned integer, rounding toward zero (vector and scalar form)	FCVTZU (vector, integer)
	Floating-point convert to unsigned fixed-point, rounding toward zero, (vector and scalar form)	FCVTZU (vector, fixed-point)

C3.5.21.2 Integer conversion

Table C3-86 shows the SIMD integer conversion instructions.

Table C3-86 SIMD integer conversion instructions

Mnemonic	Instruction	See
SCVTF	Signed integer convert to floating-point (vector and scalar form)	SCVTF (vector, integer)
	Signed fixed-point convert to floating-point (vector and scalar form)	SCVTF (vector, fixed-point)
UCVTF	Unsigned integer convert to floating-point (vector and scalar form)	UCVTF (vector, integer)
	Unsigned fixed-point convert to floating-point (vector and scalar form)	UCVTF (vector, fixed-point)

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.5.22 SIMD reduce (across vector lanes)

The SIMD reduce (across vector lanes) instructions perform arithmetic operations horizontally, that is across all lanes of the input vector. They deliver a single scalar result.

Table C3-87 shows the SIMD reduce (across vector lanes) instructions.

Table C3-87 SIMD reduce (across vector lanes) instructions

Mnemonic	Instruction	See
ADDV	Add (across vector)	ADDV
FMAXNMV	Floating-point maximum number (across vector)	FMAXNMV
FMAXV	Floating-point maximum (across vector)	FMAXV
FMINNMV	Floating-point minimum number (across vector)	FMINNMV
FMINV	Floating-point minimum (across vector)	FMINV
SADDLV	Signed add long (across vector)	SADDLV
SMAXV	Signed maximum (across vector)	SMAXV
SMINV	Signed minimum (across vector)	SMINV
UADDLV	Unsigned add long (across vector)	UADDLV
UMAXV	Unsigned maximum (across vector)	UMAXV
UMINV	Unsigned minimum (across vector)	UMINV

C3.5.23 SIMD pairwise arithmetic

The SIMD pairwise arithmetic instructions perform operations on pairs of adjacent elements and deliver a vector result.

Table C3-88 shows the SIMD pairwise arithmetic instructions.

Table C3-88 SIMD pairwise arithmetic instructions

Mnemonic	Instruction	See
ADDP	Add pairwise (vector and scalar form)	ADDP (vector) ADDP (scalar)
FADDP	Floating-point add pairwise (vector and scalar form)	FADDP (vector) FADDP (scalar)
FMAXNMP	Floating-point maximum number pairwise (vector and scalar form)	FMAXNMP (vector) FMAXNMP (scalar)
FMAXP	Floating-point maximum pairwise (vector and scalar form)	FMAXP (vector) FMAXP (scalar)
FMINNMP	Floating-point minimum number pairwise (vector and scalar form)	FMINNMP (vector) FMINNMP (scalar)
FMINP	Floating-point minimum pairwise (vector and scalar form)	FMINP (vector) FMINP (scalar)
SMAXP	Signed maximum pairwise	SMAXP

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mnemonic	Instruction	See
SMINP	Signed minimum pairwise	SMINP
UMAXP	Unsigned maximum pairwise	UMAXP
UMINP	Unsigned minimum pairwise	UMINP

C3.5.24 SIMD dot product

C3.5.24.1 Integer dot product

FEAT_DotProd provides SIMD instructions that perform the dot product of the four 8-bit subelements of the 32-bit elements of one vector with the four 8-bit subelements of a second vector. It provides two forms of the instructions, each with signed and unsigned versions:

Vector form The dot product is calculated for each element of the first vector with the corresponding element of the second vector.

Indexed form The dot product is calculated for each element of the first vector with the element of the second vector that is indicated by the index argument to the instruction.

Note

That is, a single element from the second vector is used, and the dot product is calculated between each element of the first vector and this single element from the second vector.

Table C3-89 shows the SIMD integer dot product instructions.

Table C3-89 SIMD integer dot product instructions

Mnemonic	Instruction	See
SDOT	Signed dot product (vector form)	SDOT (vector)
UDOT	Unsigned dot product (vector form)	UDOT (vector)
SDOT	Signed dot product (indexed form)	SDOT (by element)
UDOT	Unsigned dot product (indexed form)	UDOT (by element)

C3.5.25 SIMD table lookup

C3.5.25.1 Table vector lookup

Table C3-90 shows the SIMD table vector lookup instructions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-90 SIMD table vector lookup instructions

Mnemonic	Instruction	See
TBL	Table vector lookup	TBL
TBX	Table vector lookup extension	TBX

C3.5.26 SIMD complex number arithmetic instructions

FEAT_FCMA provides SIMD instructions that perform arithmetic on complex numbers held in element pairs in vector registers, where the less significant element of the pair contains the real component and the more significant element contains the imaginary component.

These instructions provide double-precision and single-precision versions. If FEAT_FP16 is implemented they also provide half-precision versions, otherwise the half-precision encodings are UNDEFINED.

Table C3-91 shows the FEAT_FCMA SIMD instructions.

Table C3-91 SIMD complex number arithmetic instructions

Mnemonic	Instruction	See
FCADD	Floating-point complex add	FCADD
FCMLA	Floating-point complex multiply accumulate (vector form)	FCMLA
FCMLA	Floating-point complex multiply accumulate (indexed form)	FCMLA (by element)

A pair of FCMLA instructions can be used to perform a complex number multiplication. This is demonstrated in Complex multiplication.

C3.5.27 SIMD multiply-accumulate

C3.5.27.1 Floating-point fused multiply-accumulate

The SIMD floating-point fused multiply-accumulate instructions multiply the vector elements in the first source SIMD&FP register by the corresponding value in the second source SIMD&FP register, and accumulate or subtract the product in the vector elements of the destination SIMD&FP register.

Table C3-92 shows these instructions.

Table C3-92 SIMD floating-point fused multiply-accumulate instructions

Mnemonic	Instruction	See
FMLA	Floating-point fused multiply-add (vector form)	FMLA (vector)
FMLAL, FMLAL2	Floating-point fused multiply-add long (vector form)	FMLAL, FMLAL2 (vector)
FMLS	Floating-point fused multiply-subtract (vector form)	FMLS (vector)
FMLSL, FMLSL2	Floating-point fused multiply-subtract long (vector form)	FMLSL, FMLSL2 (vector)

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.5.28 The Cryptographic Extension in AArch64 state

C3.5.28.1 The Armv8.0 Cryptographic Extension

The instructions provided by the OPTIONAL Armv8.0 Cryptographic Extension use the SIMD and floating-point register file. For more information about the functions they provide, see:

- *Advanced Encryption Standard (AES)*.
- *Secure Hash Algorithm (SHA)*.
- *The Galois/Counter Mode of Operation*.

Table C3-93 shows the Armv8.0 Cryptographic Extension instructions.

Table C3-93 Cryptographic Extension instructions

Mnemonic	Instruction	See
AESD	AES single round decryption	AESD
AESE	AES single round encryption	AESE
AESIMC	AES inverse mix columns	AESIMC
AESMC	AES mix columns	AESMC
PMULL	Polynomial multiply long	PMULL, PMULL2 [a]
SHA1C	SHA1 hash update (choose)	SHA1C
SHA1H	SHA1 fixed rotate	SHA1H
SHA1M	SHA1 hash update (majority)	SHA1M
SHA1P	SHA1 hash update (parity)	SHA1P
SHA1SU0	SHA1 schedule update 0	SHA1SU0
SHA1SU1	SHA1 schedule update 1	SHA1SU1
SHA256H	SHA256 hash update, part 1	SHA256H
SHA256H2	SHA256 hash update, part 2	SHA256H2
SHA256SU0	SHA256 schedule update 0	SHA256SU0
SHA256SU1	SHA256 schedule update 1	SHA256SU1

a. The Cryptographic Extension adds the variant of the instruction that operates on two 64-bit polynomials.

C3.5.28.2 Armv8.2 extensions to the Cryptographic Extension

Armv8.2 supports the following OPTIONAL extensions to the Cryptographic Extension:

- FEAT_SHA512, SHA2-512 functionality.
- FEAT_SHA3, SHA3 functionality.
- FEAT_SM3, SM3 functionality.
- FEAT_SM4, SM4 functionality.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.5.28.2.1 FEAT_SHA512, SHA2-512 functionality

FEAT_SHA512 provides instructions to accelerate the SHA-2 hash algorithm using a digest that is larger than 256 bits. The relevant standards are SHA-384, SHA-512, SHA-512|224 and SHA-512|256. These are all based on the SHA-512 computation, and therefore this set of instructions is described as the *SHA512 instructions*.

Implementation of FEAT_SHA512 requires the implementation of the SHA1 and SHA2-256 instructions from the Armv8.0 Cryptographic Extension.

Note

Implementation of FEAT_SHA512 does not require the implementation of the AES instructions, and the 64-bit polynomial variants of the `PMULL` instructions, from the Armv8.0 Cryptographic Extension.

When FEAT_SHA512 is implemented, the value of ID_AA64ISAR0_EL1.SHA2 is `0b0010`, indicating support for the SHA512 instructions.

Table C3-94 shows the FEAT_SHA512 instructions:

Table C3-94 FEAT_SHA512 instructions

Mnemonic	Instruction	See
`SHA512H`	SHA512 Hash update part 1	SHA512H
`SHA512H2`	SHA512 Hash update part 2	SHA512H2
`SHA512SU0`	SHA512 Schedule Update 0	SHA512SU0
`SHA512SU1`	SHA512 Schedule Update 1	SHA512SU1

Use of the SHA512 instructions shows an example of the use of these instructions to calculate a SHA512 hash iteration. This example code is not part of the architectural definition of these instructions.

C3.5.28.2.2 FEAT_SHA3, SHA3 functionality

FEAT_SHA3 provides instructions to accelerate the SHA-3 hash algorithm. This set of instructions is described as the *SHA3 instructions.*

Note

Implementation of FEAT_SHA3 does not require the implementation of the AES instructions, and the 64-bit polynomial variants of the `PMULL` instructions, from the Armv8.0 Cryptographic Extension.

When FEAT_SHA3 is implemented, the value of ID_AA64ISAR0_EL1.SHA3 is `0b0001`, indicating support for the SHA3 instructions.

Table C3-95 shows the FEAT_SHA3 instructions. The SHA-3 hash algorithm is based on a running digest of 1600 bytes, arranged as a five by five array of 64-bit registers. The Arm acceleration of these instructions is based on mapping the 25 64-bit values into 25 vector registers, with each 64-bit value occupying the same 64-bit element in each vector. A series of transformations is performed on these registers as part of a round of the SHA-3 hash calculation.

The SIMD nature of the vector registers means the acceleration can compute two parallel SHA3 hash calculations, where one calculation is performed using the zeroth 64-bit element of each vector, and the other calculation is performed using the first 64-bit element of each vector.

To provide acceleration where the SIMD calculation is not required, the instructions provide variants that operate only on the zeroth 64-bit elements. These are provided as a power optimization.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C3-95 FEAT_SHA3 instructions

Mnemonic	Instruction	See
BCAX	Bit Clear and Exclusive OR	BCAX
EOR3	Three-way Exclusive OR	EOR3
RAX1	Rotate and Exclusive OR	RAX1
XAR	Exclusive OR and Rotate	XAR

Use of the SHA3 instructions shows an example of the use of these instructions to calculate the combined theta, phi, rho and chi operations of a SHA3 iteration. This example code is not part of the architectural definition of these instructions.

C3.5.28.2.3 FEAT_SM3, SM3 functionality

FEAT_SM3 provides instructions to accelerate the SM3 hash algorithm, the standard Chinese hash algorithm. These are described as the *SM3 instructions*.

FEAT_SM3 can be implemented independently of any part of the Armv8.0 Cryptographic Extension, and independently of FEAT_SHA512.

Note

This means that Armv8.2 permits an implementation of the Cryptographic Extension that provides only the FEAT_SM3 functionality.

When FEAT_SM3 is implemented, the value of ID_AA64ISAR0_EL1.SM3 is 0b0001, indicating support for the SM3 instructions.

Table C3-96 shows the FEAT_SM3 instructions. The SM3 algorithm computes a digest of 256 bits, which can be held in two vector registers. The SM3 instructions include instructions to accelerate the computation of the hash and the schedule update.

Note

The SM3 instruction names refer to intermediate variables defined as part of the *SM3 Cryptographic Hash Algorithm* specification.

Table C3-96 FEAT_SM3 instructions

Mnemonic	Instruction	See
SM3SS1	SM3 SS1 calculation	SM3SS1
SM3TT1A	SM3 TT1 calculation, part A	SM3TT1A
SM3TT1B	SM3 TT1 calculation, part B	SM3TT1B
SM3TT2A	SM3 TT2 calculation, part A	SM3TT2A
SM3TT2B	SM3 TT2 calculation, part B	SM3TT2B
SM3PARTW1	SM3 PARTW calculation, part 1	SM3PARTW1
SM3PARTW2	SM3 PARTW calculation, part 1	SM3PARTW2

Use of the SM3 instructions shows an example of the use of these instructions to generate an SM3 hash. This example code is not part of the architectural definition of these instructions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C3.5.28.2.4 FEAT_SM4, SM4 functionality

FEAT_SM4 provides instruction to accelerate the SM4 encryption algorithm, the standard Chinese encryption algorithm. This set of instructions is described as the *SM4 instructions*.

FEAT_SM4 can be implemented independently of any part of the Armv8.0 Cryptographic Extension, and independently of FEAT_SHA3.

Note

This means that Armv8.2 permits an implementation of the Cryptographic Extension that provides only the FEAT_SM4 functionality.

When FEAT_SM4 is implemented, the value of ID_AA64ISAR0_EL1.SM4 is `0b0001`, indicating support for the SM4 instructions.

Table C3-97 shows the FEAT_SM4 instructions. The SM4 algorithm is 128-bit wide block cipher. The `SM4E` instruction accelerates a single round of encryption or decryption, and the `SM4EKEY` instruction accelerates a single round of key generation:

Table C3-97 FEAT_SM4 instructions

Mnemonic	Instruction	See
`SM4E`	SM4 Encrypt	SM4E
`SM4EKEY`	SM4 Key	SM4EKEY

Use of the SM4 instructions shows an example of the use of these instructions to perform SM4 encryption and decryption. This example code is not part of the architectural definition of these instructions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter C4
A64 Instruction Set Encoding

This chapter describes the encoding of the A64 instruction set. It contains the following section:

- A64 instruction set encoding.

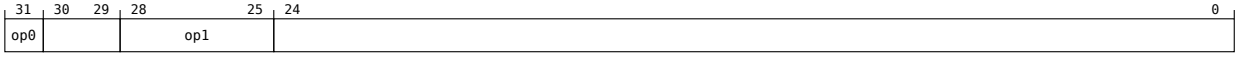
In this chapter:

- In the decode tables, an entry of - for a field value means the value of the field does not affect the decoding.
- In the decode diagrams, a shaded field indicates that the bits in that field are not used in that level of decode.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

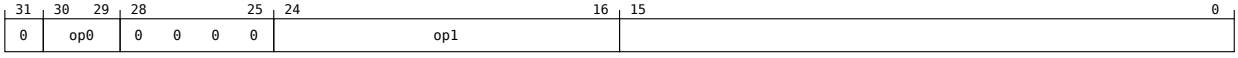
C4.1 A64 instruction set encoding



op0	op1	Instruction details
0	0000	Reserved
0	0001	UNALLOCATED
0	001x	UNALLOCATED
1	00xx	UNALLOCATED
	100x	Data Processing – Immediate
	101x	Branches, Exception Generating and System instructions
	x101	Data Processing – Register
	x111	Data Processing – Scalar Floating-Point and Advanced SIMD
	x1x0	Loads and Stores

C4.1.1 Reserved

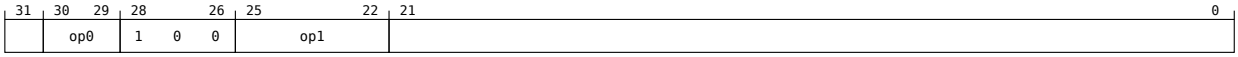
The encodings in this section are decoded from A64 instruction set encoding.



op0	op1	Instruction details
00	000000000	UDF
00	!= 000000000	UNALLOCATED
!= 00		UNALLOCATED

C4.1.2 Data Processing – Immediate

The encodings in this section are decoded from A64 instruction set encoding.



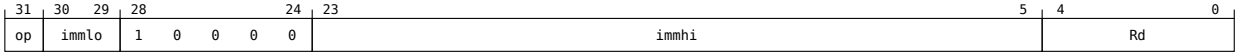
op0	op1	Instruction details
11	111x	UNALLOCATED

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	Instruction details
	00xx	PC-rel. addressing
	010x	Add/subtract (immediate)
	011x	UNALLOCATED
	100x	Logical (immediate)
	101x	Move wide (immediate)
	110x	Bitfield
!= 11	111x	Extract

C4.1.2.1 PC-rel. addressing

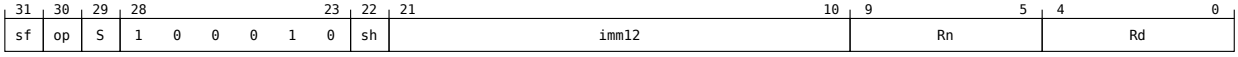
The encodings in this section are decoded from Data Processing – Immediate.



op	Instruction Details
0	ADR
1	ADRP

C4.1.2.2 Add/subtract (immediate)

The encodings in this section are decoded from Data Processing – Immediate.

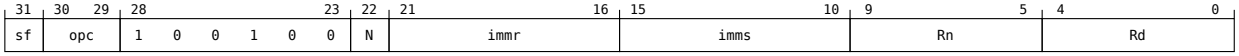


sf	op	S	Instruction Details
0	0	0	ADD (immediate) — 32-bit
0	0	1	ADDS (immediate) — 32-bit
0	1	0	SUB (immediate) — 32-bit
0	1	1	SUBS (immediate) — 32-bit
1	0	0	ADD (immediate) — 64-bit
1	0	1	ADDS (immediate) — 64-bit
1	1	0	SUB (immediate) — 64-bit
1	1	1	SUBS (immediate) — 64-bit

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C4.1.2.3 Logical (immediate)

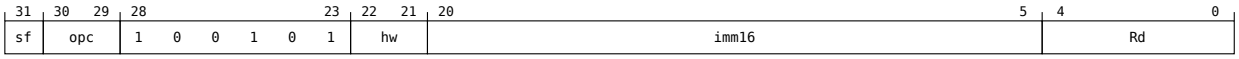
The encodings in this section are decoded from Data Processing – Immediate.



sf	opc	N	Instruction Details
0		1	UNALLOCATED
0	00	0	AND (immediate) — 32-bit
0	01	0	ORR (immediate) — 32-bit
0	10	0	EOR (immediate) — 32-bit
0	11	0	ANDS (immediate) — 32-bit
1	00		AND (immediate) — 64-bit
1	01		ORR (immediate) — 64-bit
1	10		EOR (immediate) — 64-bit
1	11		ANDS (immediate) — 64-bit

C4.1.2.4 Move wide (immediate)

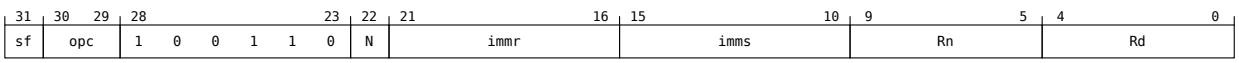
The encodings in this section are decoded from Data Processing – Immediate.



sf	opc	hw	Instruction Details
	01	0x	UNALLOCATED
0		1x	UNALLOCATED
0	00	0x	MOVN — 32-bit
0	10	0x	MOVZ — 32-bit
0	11	0x	MOVK — 32-bit
1	00		MOVN — 64-bit
1	01	1x	UNALLOCATED
1	10		MOVZ — 64-bit
1	11		MOVK — 64-bit

C4.1.2.5 Bitfield

The encodings in this section are decoded from Data Processing – Immediate.

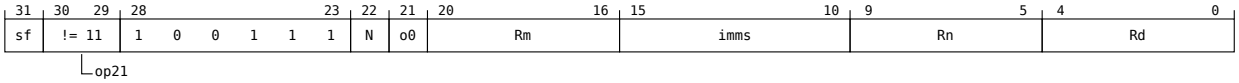


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sf	opc	N	Instruction Details
0		1	UNALLOCATED
0	00	0	SBFM — 32-bit
0	01	0	BFM — 32-bit
0	10	0	UBFM — 32-bit
0	11	0	UNALLOCATED
1		0	UNALLOCATED
1	00	1	SBFM — 64-bit
1	01	1	BFM — 64-bit
1	10	1	UBFM — 64-bit
1	11	1	UNALLOCATED

C4.1.2.6 Extract

The encodings in this section are decoded from Data Processing – Immediate.

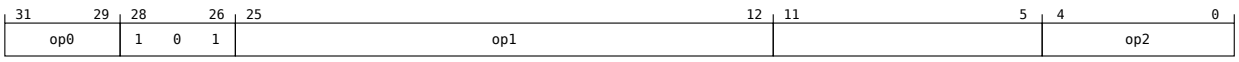


The following constraints also apply to this encoding: op21 != ’11’

sf	op21	N	o0	imms	Instruction Details
	00		1		UNALLOCATED
	01				UNALLOCATED
	10				UNALLOCATED
0	00	0	0	0xxxxx	EXTR — 32-bit
0	00	0	0	1xxxxx	UNALLOCATED
0	00	1	0		UNALLOCATED
1	00	0	0		UNALLOCATED
1	00	1	0		EXTR — 64-bit

C4.1.3 Branches, Exception Generating and System instructions

The encodings in this section are decoded from A64 instruction set encoding.

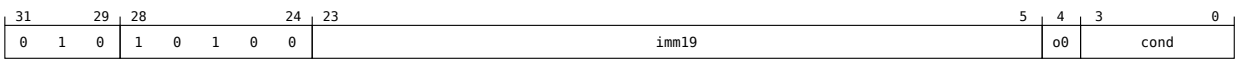


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	Instruction details
010	00xxxxxxxxxxx		Conditional branch (immediate)
010	01xxxxxxxxxxx		UNALLOCATED
010	1xxxxxxxxxxxx		UNALLOCATED
110	00xxxxxxxxxxx		Exception generation
110	010000000x0x1x		UNALLOCATED
110	01000000100x1x		UNALLOCATED
110	01000000110010	11111	Hints
110	01000000110010	!= 11111	UNALLOCATED
110	01000000110011		Barriers
110	0100000011011x		UNALLOCATED
110	01000001xx0x1x		UNALLOCATED
110	0100000xxx000x		UNALLOCATED
110	0100000xxx0100		PSTATE
110	0100000xxx0101		UNALLOCATED
110	0100000xxx1xxx		UNALLOCATED
110	0100100xxxxxxx		UNALLOCATED
110	0100x01xxxxxxx		System instructions
110	0100x1xxxxxxxx		System register move
110	0101xxxxxxxxx		UNALLOCATED
110	011xxxxxxxxxx		UNALLOCATED
110	1xxxxxxxxxxxx		Unconditional branch (register)
x00			Unconditional branch (immediate)
x01	0xxxxxxxxxxxx		Compare and branch (immediate)
x01	1xxxxxxxxxxxx		Test and branch (immediate)
x11			UNALLOCATED

C4.1.3.1 Conditional branch (immediate)

The encodings in this section are decoded from Branches, Exception Generating and System instructions.

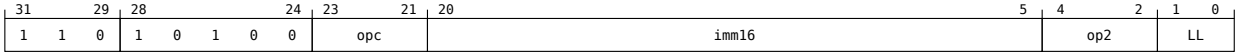


o0	Instruction Details
0	B.cond
1	UNALLOCATED

C4.1.3.2 Exception generation

The encodings in this section are decoded from Branches, Exception Generating and System instructions.

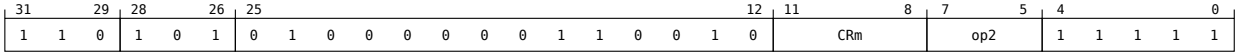
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential



opc	op2	LL	Instruction Details
	!= 000		UNALLOCATED
x11	000		UNALLOCATED
000	000	00	UNALLOCATED
000	000	01	SVC
000	000	10	HVC
000	000	11	SMC
001	000	00	BRK
001	000	!= 00	UNALLOCATED
010	000	00	HLT
010	000	!= 00	UNALLOCATED
1x0	000		UNALLOCATED
101	000	00	UNALLOCATED
101	000	01	DCPS1
101	000	10	DCPS2
101	000	11	DCPS3

C4.1.3.3 Hints

The encodings in this section are decoded from Branches, Exception Generating and System instructions.



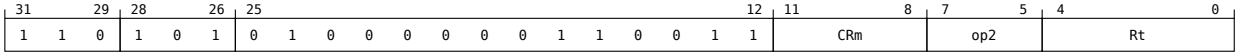
CRm	op2	Instruction Details	Feature
		HINT	-
0000	000	NOP	-
0000	001	YIELD	-
0000	010	WFE	-
0000	011	WFI	-
0000	100	SEV	-
0000	101	SEVL	-
0000	110	DGH	FEAT_DGH
0000	111	XPACD, XPACI, XPACLRI	FEAT_PAuth
0001	000	PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA — PACIA1716	FEAT_PAuth
0001	010	PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB — PACIB1716	FEAT_PAuth

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CRm	op2	Instruction Details	Feature
0001	100	AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA — AUTIA1716	FEAT_PAuth
0001	110	AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB — AUTIB1716	FEAT_PAuth
0010	000	ESB	FEAT_RAS
0010	010	TSB	FEAT_TRF
0010	100	CSDB	-
0011	000	PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA — PACIAZ	FEAT_PAuth
0011	001	PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA — PACIASP	FEAT_PAuth
0011	010	PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB — PACIBZ	FEAT_PAuth
0011	011	PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB — PACIBSP	FEAT_PAuth
0011	100	AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA — AUTIAZ	FEAT_PAuth
0011	101	AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA — AUTIASP	FEAT_PAuth
0011	110	AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB — AUTIBZ	FEAT_PAuth
0011	111	AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB — AUTIBSP	FEAT_PAuth

C4.1.3.4 Barriers

The encodings in this section are decoded from Branches, Exception Generating and System instructions.

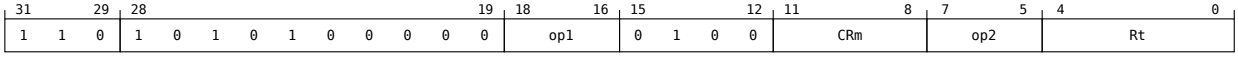


op2	Rt	Instruction Details	Feature
	!= 11111	UNALLOCATED	-
00x	11111	UNALLOCATED	-
010	11111	CLREX	-
011	11111	UNALLOCATED	-
100	11111	DSB	-
101	11111	DMB	-
110	11111	ISB	-
111	11111	SB	FEAT_SB

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C4.1.3.5 PSTATE

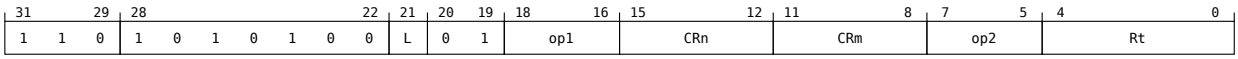
The encodings in this section are decoded from Branches, Exception Generating and System instructions.



op1:op2	Rt	Instruction Details	Feature
	0xxxx	UNALLOCATED	-
	10xxx	UNALLOCATED	-
	110xx	UNALLOCATED	-
	1110x	UNALLOCATED	-
	11110	UNALLOCATED	-
000001	11111	UNALLOCATED	-
000010	11111	UNALLOCATED	-
000000	11111	CFINV	FEAT_FlagM
!= 00000x	11111	MSR (immediate)	-

C4.1.3.6 System instructions

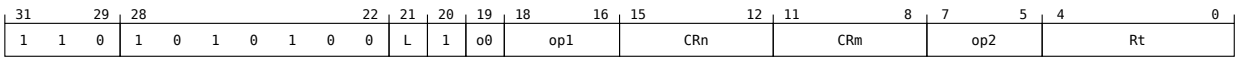
The encodings in this section are decoded from Branches, Exception Generating and System instructions.



L	Instruction Details
0	SYS
1	SYSL

C4.1.3.7 System register move

The encodings in this section are decoded from Branches, Exception Generating and System instructions.

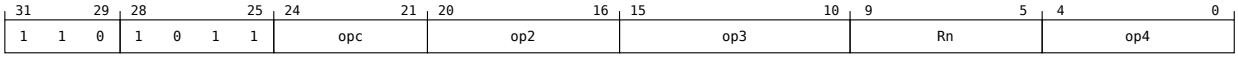


L	Instruction Details
0	MSR (register)
1	MRS

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C4.1.3.8 Unconditional branch (register)

The encodings in this section are decoded from Branches, Exception Generating and System instructions.



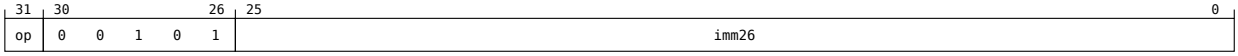
opc	op2	op3	Rn	op4	Instruction Details	Feature
	11111	0001xx			UNALLOCATED	-
	11111	001xxx			UNALLOCATED	-
	11111	01xxxx			UNALLOCATED	-
	11111	1xxxxx			UNALLOCATED	-
	!= 11111				UNALLOCATED	-
00x0	11111	000000		!= 00000	UNALLOCATED	-
00x0	11111	000001			UNALLOCATED	-
00x0	11111	00001x		!= 11111	UNALLOCATED	-
0000	11111	000000		00000	BR	-
0000	11111	000010		11111	BRAA, BRAAZ, BRAB, BRABZ — Key A, zero modifier	FEAT_PAuth
0000	11111	000011		11111	BRAA, BRAAZ, BRAB, BRABZ — Key B, zero modifier	FEAT_PAuth
0001	11111	000000		00000	BLR	-
0001	11111	000000		!= 00000	UNALLOCATED	-
0001	11111	000001			UNALLOCATED	-
0001	11111	00001x		!= 11111	UNALLOCATED	-
0001	11111	000010		11111	BLRAA, BLRAAZ, BLRAB, BLRABZ — Key A, zero modifier	FEAT_PAuth
0001	11111	000011		11111	BLRAA, BLRAAZ, BLRAB, BLRABZ — Key B, zero modifier	FEAT_PAuth
0010	11111	000000		00000	RET	-
0010	11111	00001x	!= 11111	11111	UNALLOCATED	-
0010	11111	000010	11111	11111	RETAA, RETAB — RETAA	FEAT_PAuth
0010	11111	000011	11111	11111	RETAA, RETAB — RETAB	FEAT_PAuth
0011	11111	0000xx			UNALLOCATED	-
010x	11111	0000xx	!= 11111		UNALLOCATED	-
010x	11111	000000	11111	!= 00000	UNALLOCATED	-
010x	11111	000001	11111		UNALLOCATED	-
010x	11111	00001x	11111	!= 11111	UNALLOCATED	-
0100	11111	000000	11111	00000	ERET	-
0100	11111	000010	11111	11111	ERETAA, ERETAB — ERETAA	FEAT_PAuth
0100	11111	000011	11111	11111	ERETAA, ERETAB — ERETAB	FEAT_PAuth
0101	11111	000000	11111	00000	DRPS	-
0101	11111	00001x	11111	11111	UNALLOCATED	-

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	op2	op3	Rn	op4	Instruction Details	Feature
011x	11111	0000xx			UNALLOCATED	-
100x	11111	00000x			UNALLOCATED	-
1000	11111	000010			BRAA, BRAAZ, BRAB, BRABZ — Key A, register modifier	FEAT_PAuth
1000	11111	000011			BRAA, BRAAZ, BRAB, BRABZ — Key B, register modifier	FEAT_PAuth
1001	11111	000010			BLRAA, BLRAAZ, BLRAB, BLRABZ — Key A, register modifier	FEAT_PAuth
1001	11111	000011			BLRAA, BLRAAZ, BLRAB, BLRABZ — Key B, register modifier	FEAT_PAuth
101x	11111	0000xx			UNALLOCATED	-
11xx	11111	0000xx			UNALLOCATED	-

C4.1.3.9 Unconditional branch (immediate)

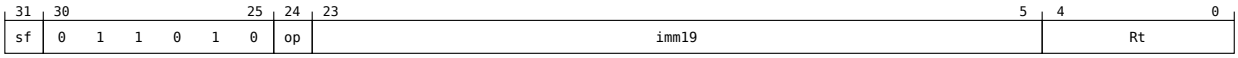
The encodings in this section are decoded from Branches, Exception Generating and System instructions.



op	Instruction Details
0	B
1	BL

C4.1.3.10 Compare and branch (immediate)

The encodings in this section are decoded from Branches, Exception Generating and System instructions.

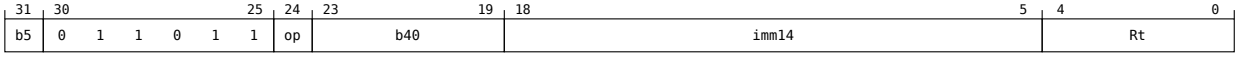


sf	op	Instruction Details
0	0	CBZ — 32-bit
0	1	CBNZ — 32-bit
1	0	CBZ — 64-bit
1	1	CBNZ — 64-bit

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C4.1.3.11 Test and branch (immediate)

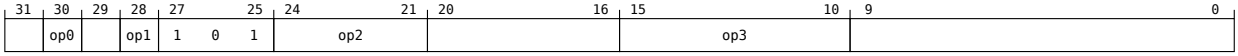
The encodings in this section are decoded from Branches, Exception Generating and System instructions.



op	Instruction Details
0	TBZ
1	TBNZ

C4.1.4 Data Processing – Register

The encodings in this section are decoded from A64 instruction set encoding.

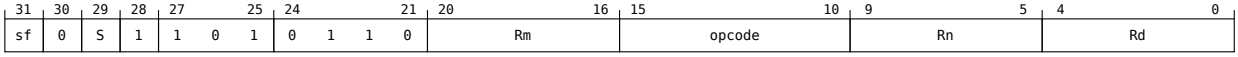


op0	op1	op2	op3	Instruction details
0	1	0110		Data-processing (2 source)
1	1	0110		Data-processing (1 source)
	0	0xxx		Logical (shifted register)
	0	1xx0		Add/subtract (shifted register)
	0	1xx1		Add/subtract (extended register)
	1	0000	000000	Add/subtract (with carry)
	1	0000	100000	UNALLOCATED
	1	0000	x00001	Rotate right into flags
	1	0000	x1000x	UNALLOCATED
	1	0000	xx0010	Evaluate into flags
	1	0000	xx0011	UNALLOCATED
	1	0000	xx01xx	UNALLOCATED
	1	0000	xx1xxx	UNALLOCATED
	1	0010	xxxx0x	Conditional compare (register)
	1	0010	xxxx1x	Conditional compare (immediate)
	1	0100		Conditional select
	1	0xx1		UNALLOCATED
	1	1xxx		Data-processing (3 source)

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C4.1.4.1 Data-processing (2 source)

The encodings in this section are decoded from Data Processing – Register.



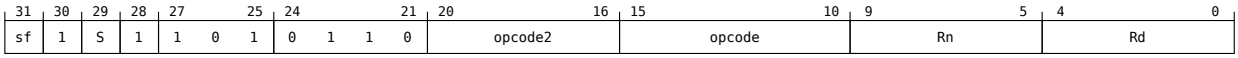
sf	S	opcode	Instruction Details	Feature
	0	00011x	UNALLOCATED	-
	0	011xxx	UNALLOCATED	-
	0	1xxxxx	UNALLOCATED	-
	1		UNALLOCATED	-
0	0	000x0x	UNALLOCATED	-
0	0	000010	UDIV — 32-bit	-
0	0	000011	SDIV — 32-bit	-
0	0	001000	LSLV — 32-bit	-
0	0	001001	LSRV — 32-bit	-
0	0	001010	ASRV — 32-bit	-
0	0	001011	RORV — 32-bit	-
0	0	0011xx	UNALLOCATED	-
0	0	010x11	UNALLOCATED	-
0	0	010000	CRC32B, CRC32H, CRC32W, CRC32X — CRC32B	FEAT_CRC32
0	0	010001	CRC32B, CRC32H, CRC32W, CRC32X — CRC32H	FEAT_CRC32
0	0	010010	CRC32B, CRC32H, CRC32W, CRC32X — CRC32W	FEAT_CRC32
0	0	010100	CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CB	FEAT_CRC32
0	0	010101	CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CH	FEAT_CRC32
0	0	010110	CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CW	FEAT_CRC32
1	0	0x0x01	UNALLOCATED	-
1	0	000x00	UNALLOCATED	-
1	0	000010	UDIV — 64-bit	-
1	0	000011	SDIV — 64-bit	-
1	0	001000	LSLV — 64-bit	-
1	0	001001	LSRV — 64-bit	-
1	0	001010	ASRV — 64-bit	-
1	0	001011	RORV — 64-bit	-
1	0	001100	PACGA	FEAT_PAuth
1	0	001101	UNALLOCATED	-
1	0	00111x	UNALLOCATED	-
1	0	010xx0	UNALLOCATED	-

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sf	S	opcode	Instruction Details	Feature
1	0	010011	CRC32B, CRC32H, CRC32W, CRC32X — CRC32X	FEAT_CRC32
1	0	010111	CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CX	FEAT_CRC32

C4.1.4.2 Data-processing (1 source)

The encodings in this section are decoded from Data Processing – Register.



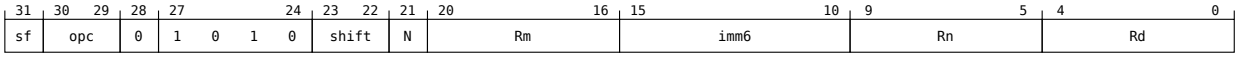
sf	S	opcode2	opcode	Rn	Instruction Details	Feature
	0	0000x	1xxxxx		UNALLOCATED	-
	0	00000	000110		UNALLOCATED	-
	0	00000	001xxx		UNALLOCATED	-
	0	00000	01xxxx		UNALLOCATED	-
	0	0001x			UNALLOCATED	-
	0	001xx			UNALLOCATED	-
	0	01xxx			UNALLOCATED	-
	0	1xxxx			UNALLOCATED	-
	1				UNALLOCATED	-
0	0	00000	000x11		UNALLOCATED	-
0	0	00000	000000		RBIT — 32-bit	-
0	0	00000	000001		REV16 — 32-bit	-
0	0	00000	000010		REV — 32-bit	-
0	0	00000	000100		CLZ — 32-bit	-
0	0	00000	000101		CLS — 32-bit	-
0	0	00001	0xxxxx		UNALLOCATED	-
1	0	00000	000000		RBIT — 64-bit	-
1	0	00000	000001		REV16 — 64-bit	-
1	0	00000	000010		REV32	-
1	0	00000	000011		REV — 64-bit	-
1	0	00000	000100		CLZ — 64-bit	-
1	0	00000	000101		CLS — 64-bit	-
1	0	00000	000111		UNALLOCATED	-
1	0	00001	000000		PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA — PACIA	FEAT_PAuth
1	0	00001	000001		PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB — PACIB	FEAT_PAuth
1	0	00001	000010		PACDA, PACDZA — PACDA	FEAT_PAuth

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sf	S	opcode2	opcode	Rn	Instruction Details	Feature
1	0	00001	000011		PACDB, PACDZB — PACDB	FEAT_PAuth
1	0	00001	000100		AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA — AUTIA	FEAT_PAuth
1	0	00001	000101		AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB — AUTIB	FEAT_PAuth
1	0	00001	000110		AUTDA, AUTDZA — AUTDA	FEAT_PAuth
1	0	00001	000111		AUTDB, AUTDZB — AUTDB	FEAT_PAuth
1	0	00001	001xxx	!= 11111	UNALLOCATED	-
1	0	00001	001000	11111	PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA — PACIZA	FEAT_PAuth
1	0	00001	001001	11111	PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB — PACIZB	FEAT_PAuth
1	0	00001	001010	11111	PACDA, PACDZA — PACDZA	FEAT_PAuth
1	0	00001	001011	11111	PACDB, PACDZB — PACDZB	FEAT_PAuth
1	0	00001	001100	11111	AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA — AUTIZA	FEAT_PAuth
1	0	00001	001101	11111	AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB — AUTIZB	FEAT_PAuth
1	0	00001	001110	11111	AUTDA, AUTDZA — AUTDZA	FEAT_PAuth
1	0	00001	001111	11111	AUTDB, AUTDZB — AUTDZB	FEAT_PAuth
1	0	00001	01000x	!= 11111	UNALLOCATED	-
1	0	00001	010000	11111	XPACD, XPACI, XPACLRI — XPACI	FEAT_PAuth
1	0	00001	010001	11111	XPACD, XPACI, XPACLRI — XPACD	FEAT_PAuth
1	0	00001	01001x		UNALLOCATED	-
1	0	00001	0101xx		UNALLOCATED	-
1	0	00001	011xxx		UNALLOCATED	-

C4.1.4.3 Logical (shifted register)

The encodings in this section are decoded from Data Processing – Register.



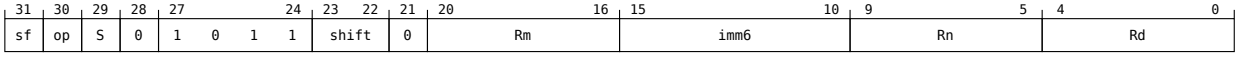
sf	opc	N	Instruction Details
0	00	0	AND (shifted register) — 32-bit
0	00	1	BIC (shifted register) — 32-bit
0	01	0	ORR (shifted register) — 32-bit
0	01	1	ORN (shifted register) — 32-bit
0	10	0	EOR (shifted register) — 32-bit
0	10	1	EON (shifted register) — 32-bit
0	11	0	ANDS (shifted register) — 32-bit

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sf	opc	N	Instruction Details
0	11	1	BICS (shifted register) — 32-bit
1	00	0	AND (shifted register) — 64-bit
1	00	1	BIC (shifted register) — 64-bit
1	01	0	ORR (shifted register) — 64-bit
1	01	1	ORN (shifted register) — 64-bit
1	10	0	EOR (shifted register) — 64-bit
1	10	1	EON (shifted register) — 64-bit
1	11	0	ANDS (shifted register) — 64-bit
1	11	1	BICS (shifted register) — 64-bit

C4.1.4.4 Add/subtract (shifted register)

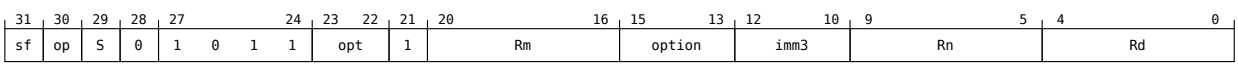
The encodings in this section are decoded from Data Processing – Register.



sf	op	S	Instruction Details
0	0	0	ADD (shifted register) — 32-bit
0	0	1	ADDS (shifted register) — 32-bit
0	1	0	SUB (shifted register) — 32-bit
0	1	1	SUBS (shifted register) — 32-bit
1	0	0	ADD (shifted register) — 64-bit
1	0	1	ADDS (shifted register) — 64-bit
1	1	0	SUB (shifted register) — 64-bit
1	1	1	SUBS (shifted register) — 64-bit

C4.1.4.5 Add/subtract (extended register)

The encodings in this section are decoded from Data Processing – Register.



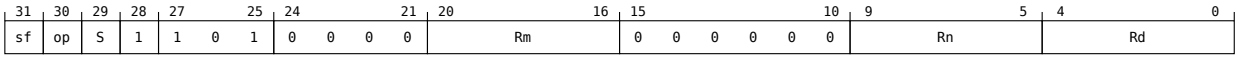
sf	op	S	opt	Instruction Details
			!= 00	UNALLOCATED
0	0	0	00	ADD (extended register) — 32-bit
0	0	1	00	ADDS (extended register) — 32-bit
0	1	0	00	SUB (extended register) — 32-bit

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sf	op	S	opt	Instruction Details
0	1	1	00	SUBS (extended register) — 32-bit
1	0	0	00	ADD (extended register) — 64-bit
1	0	1	00	ADDS (extended register) — 64-bit
1	1	0	00	SUB (extended register) — 64-bit
1	1	1	00	SUBS (extended register) — 64-bit

C4.1.4.6 Add/subtract (with carry)

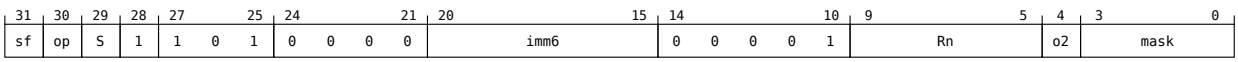
The encodings in this section are decoded from Data Processing – Register.



sf	op	S	Instruction Details
0	0	0	ADC — 32-bit
0	0	1	ADCS — 32-bit
0	1	0	SBC — 32-bit
0	1	1	SBCS — 32-bit
1	0	0	ADC — 64-bit
1	0	1	ADCS — 64-bit
1	1	0	SBC — 64-bit
1	1	1	SBCS — 64-bit

C4.1.4.7 Rotate right into flags

The encodings in this section are decoded from Data Processing – Register.

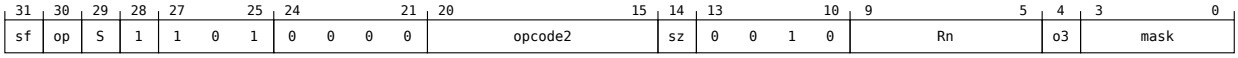


sf	op	S	o2	Instruction Details	Feature
0				UNALLOCATED	-
1	0	0		UNALLOCATED	-
1	0	1	0	RMIF	FEAT_FlagM
1	0	1	1	UNALLOCATED	-
1	1			UNALLOCATED	-

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C4.1.4.8 Evaluate into flags

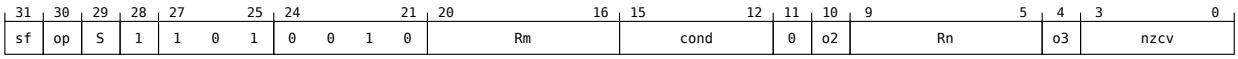
The encodings in this section are decoded from Data Processing – Register.



sf	op	S	opcode2	sz	o3	mask	Instruction Details	Feature
0	0	0					UNALLOCATED	-
0	0	1	000000		0	!= 1101	UNALLOCATED	-
0	0	1	000000		1		UNALLOCATED	-
0	0	1	000000	0	0	1101	SETF8, SETF16 — SETF8	FEAT_FlagM
0	0	1	000000	1	0	1101	SETF8, SETF16 — SETF16	FEAT_FlagM
0	0	1	!= 000000				UNALLOCATED	-
0	1						UNALLOCATED	-
1							UNALLOCATED	-

C4.1.4.9 Conditional compare (register)

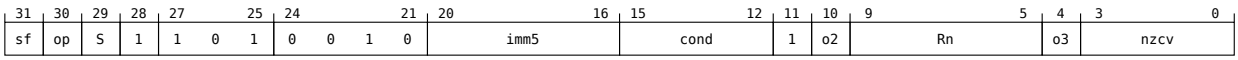
The encodings in this section are decoded from Data Processing – Register.



sf	op	S	o2	o3	Instruction Details
		0			UNALLOCATED
		1	0	1	UNALLOCATED
		1	1		UNALLOCATED
0	0	1	0	0	CCMN (register) — 32-bit
0	1	1	0	0	CCMP (register) — 32-bit
1	0	1	0	0	CCMN (register) — 64-bit
1	1	1	0	0	CCMP (register) — 64-bit

C4.1.4.10 Conditional compare (immediate)

The encodings in this section are decoded from Data Processing – Register.

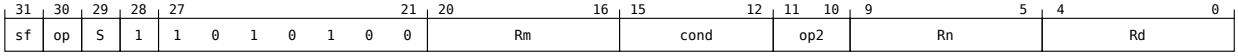


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sf	op	S	o2	o3	Instruction Details
		0			UNALLOCATED
		1	0	1	UNALLOCATED
		1	1		UNALLOCATED
0	0	1	0	0	CCMN (immediate) — 32-bit
0	1	1	0	0	CCMP (immediate) — 32-bit
1	0	1	0	0	CCMN (immediate) — 64-bit
1	1	1	0	0	CCMP (immediate) — 64-bit

C4.1.4.11 Conditional select

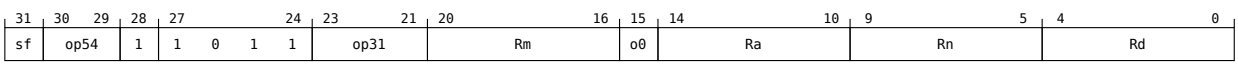
The encodings in this section are decoded from Data Processing – Register.



sf	op	S	op2	Instruction Details
		0	1x	UNALLOCATED
		1		UNALLOCATED
0	0	0	00	CSEL — 32-bit
0	0	0	01	CSINC — 32-bit
0	1	0	00	CSINV — 32-bit
0	1	0	01	CSNEG — 32-bit
1	0	0	00	CSEL — 64-bit
1	0	0	01	CSINC — 64-bit
1	1	0	00	CSINV — 64-bit
1	1	0	01	CSNEG — 64-bit

C4.1.4.12 Data-processing (3 source)

The encodings in this section are decoded from Data Processing – Register.



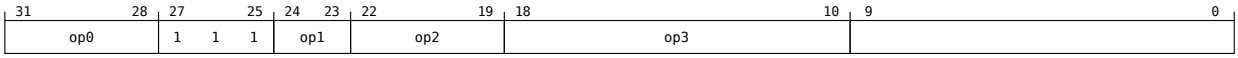
sf	op54	op31	o0	Instruction Details
	00	100		UNALLOCATED
	!= 00			UNALLOCATED
0	00	x01		UNALLOCATED
0	00	x1x		UNALLOCATED

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sf	op54	op31	o0	Instruction Details
0	00	000	0	MADD — 32-bit
0	00	000	1	MSUB — 32-bit
1	00	x10	1	UNALLOCATED
1	00	x11		UNALLOCATED
1	00	000	0	MADD — 64-bit
1	00	000	1	MSUB — 64-bit
1	00	001	0	SMADDL
1	00	001	1	SMSUBL
1	00	010	0	SMULH
1	00	101	0	UMADDL
1	00	101	1	UMSUBL
1	00	110	0	UMULH

C4.1.5 Data Processing – Scalar Floating-Point and Advanced SIMD

The encodings in this section are decoded from A64 instruction set encoding.



op0	op1	op2	op3	Instruction details	Feature
00x0	0x	x101	00xxxxx10	UNALLOCATED	-
00x0	11		xxxxxxxx1	UNALLOCATED	-
0100	0x	x101	00xxxxx10	Cryptographic AES	-
0101	0x	x0xx	xxx0xxx00	Cryptographic three-register SHA	-
0101	0x	x0xx	xxx0xxx10	UNALLOCATED	-
0101	0x	x0xx	xxx1xxxx0	UNALLOCATED	-
0101	0x	x101	00xxxxx10	Cryptographic two-register SHA	-
0111	0x	x0xx	xxxxxxxx0	UNALLOCATED	-
011x	0x	x101	00xxxxx10	UNALLOCATED	-
01x1	00	00xx	xxx0xxxx1	Advanced SIMD scalar copy	-
01x1	01	00xx	xxx0xxxx1	UNALLOCATED	-
01x1	0x	0111	00xxxxx10	UNALLOCATED	-
01x1	0x	10xx	xxx00xxx1	Advanced SIMD scalar three same FP16	-
01x1	0x	10xx	xxx01xxx1	UNALLOCATED	-
01x1	0x	1111	00xxxxx10	Advanced SIMD scalar two-register miscellaneous FP16	-
01x1	0x	x0xx	xxx1xxxx1	Advanced SIMD scalar three same extra	-
01x1	0x	x100	00xxxxx10	Advanced SIMD scalar two-register miscellaneous	-

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

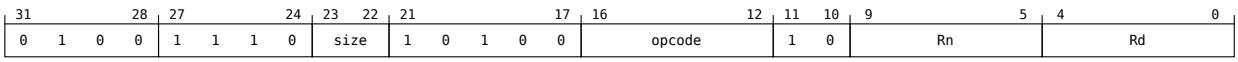
op0	op1	op2	op3	Instruction details	Feature
01x1	0x	x110	00xxxxx10	Advanced SIMD scalar pairwise	-
01x1	0x	x1xx	01xxxxx10	UNALLOCATED	-
01x1	0x	x1xx	1xxxxxx10	UNALLOCATED	-
01x1	0x	x1xx	xxxxxxx00	Advanced SIMD scalar three different	-
01x1	0x	x1xx	xxxxxxxx1	Advanced SIMD scalar three same	-
01x1	10		xxxxxxxx1	Advanced SIMD scalar shift by immediate	-
01x1	1x		xxxxxxxx0	Advanced SIMD scalar x indexed element	-
01xx	11		xxxxxxxx1	UNALLOCATED	-
0x00	0x	x0xx	xxx0xxx00	Advanced SIMD table lookup	-
0x00	0x	x0xx	xxx0xxx10	Advanced SIMD permute	-
0x10	0x	x0xx	xxx0xxxx0	Advanced SIMD extract	-
0xx0	00	00xx	xxx0xxxx1	Advanced SIMD copy	-
0xx0	01	00xx	xxx0xxxx1	UNALLOCATED	-
0xx0	0x	0111	00xxxxx10	UNALLOCATED	-
0xx0	0x	10xx	xxx00xxx1	Advanced SIMD three same (FP16)	-
0xx0	0x	10xx	xxx01xxx1	UNALLOCATED	-
0xx0	0x	1111	00xxxxx10	Advanced SIMD two-register miscellaneous (FP16)	-
0xx0	0x	x0xx	xxx1xxxx0	UNALLOCATED	-
0xx0	0x	x0xx	xxx1xxxx1	Advanced SIMD three-register extension	-
0xx0	0x	x100	00xxxxx10	Advanced SIMD two-register miscellaneous	-
0xx0	0x	x110	00xxxxx10	Advanced SIMD across lanes	-
0xx0	0x	x1xx	01xxxxx10	UNALLOCATED	-
0xx0	0x	x1xx	1xxxxxx10	UNALLOCATED	-
0xx0	0x	x1xx	xxxxxxx00	Advanced SIMD three different	-
0xx0	0x	x1xx	xxxxxxxx1	Advanced SIMD three same	-
0xx0	10	0000	xxxxxxxx1	Advanced SIMD modified immediate	-
0xx0	10	!= 0000	xxxxxxxx1	Advanced SIMD shift by immediate	-
0xx0	1x		xxxxxxxx0	Advanced SIMD vector x indexed element	-
10x0				UNALLOCATED	-
1100	00	0xxx	xxx1xxxxx	UNALLOCATED	-
1100	00	10xx	xxx10xxxx	Cryptographic three-register, imm2	-
1100	00	10xx	xxx11xxxx	UNALLOCATED	-
1100	00	11xx	xxx1x00xx	Cryptographic three-register SHA 512	-
1100	00	11xx	xxx1x01xx	UNALLOCATED	-
1100	00	11xx	xxx1x1xxx	UNALLOCATED	-
1100	00		xxx0xxxxx	Cryptographic four-register	-
1100	01	00xx		XAR	FEAT_SHA3
1100	01	1000	0000xxxxx	UNALLOCATED	-

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	op3	Instruction details	Feature
1100	01	1000	0001000xx	Cryptographic two-register SHA 512	-
1100	01	1000	0001100xx	UNALLOCATED	-
1100	01	1000	0001x01xx	UNALLOCATED	-
1100	01	1000	0001x1xxx	UNALLOCATED	-
1100	01	1000	001xxxxxx	UNALLOCATED	-
1100	01	1000	01xxxxxxx	UNALLOCATED	-
1100	01	1000	1xxxxxxxx	UNALLOCATED	-
1100	01	1001		UNALLOCATED	-
1100	01	101x		UNALLOCATED	-
1100	01	x1xx		UNALLOCATED	-
1100	1x			UNALLOCATED	-
1110				UNALLOCATED	-
11x1				UNALLOCATED	-
x0x1	0x	x0xx		Conversion between floating-point and fixed-point	-
x0x1	0x	x1xx	xxx000000	Conversion between floating-point and integer	-
x0x1	0x	x1xx	xxx100000	UNALLOCATED	-
x0x1	0x	x1xx	xxxx10000	Floating-point data-processing (1 source)	-
x0x1	0x	x1xx	xxxxx1000	Floating-point compare	-
x0x1	0x	x1xx	xxxxxx100	Floating-point immediate	-
x0x1	0x	x1xx	xxxxxxx01	Floating-point conditional compare	-
x0x1	0x	x1xx	xxxxxxx10	Floating-point data-processing (2 source)	-
x0x1	0x	x1xx	xxxxxxx11	Floating-point conditional select	-
x0x1	1x			Floating-point data-processing (3 source)	-

C4.1.5.1 Cryptographic AES

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.



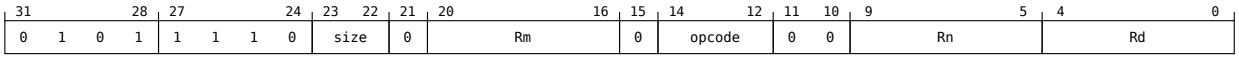
size	opcode	Instruction Details	Feature
00	000xx	UNALLOCATED	-
00	00100	AESE	FEAT_AES
00	00101	AESD	FEAT_AES
00	00110	AESMC	FEAT_AES
00	00111	AESIMC	FEAT_AES
00	01xxx	UNALLOCATED	-

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	opcode	Instruction Details	Feature
00	1xxxx	UNALLOCATED	-
!= 00		UNALLOCATED	-

C4.1.5.2 Cryptographic three-register SHA

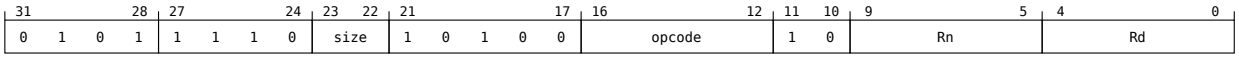
The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.



size	opcode	Instruction Details	Feature
00	000	SHA1C	FEAT_SHA1
00	001	SHA1P	FEAT_SHA1
00	010	SHA1M	FEAT_SHA1
00	011	SHA1SU0	FEAT_SHA1
00	100	SHA256H	FEAT_SHA256
00	101	SHA256H2	FEAT_SHA256
00	110	SHA256SU1	FEAT_SHA256
00	111	UNALLOCATED	-
!= 00		UNALLOCATED	-

C4.1.5.3 Cryptographic two-register SHA

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

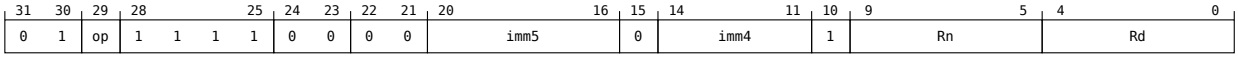


size	opcode	Instruction Details	Feature
00	00000	SHA1H	FEAT_SHA1
00	00001	SHA1SU1	FEAT_SHA1
00	00010	SHA256SU0	FEAT_SHA256
00	00011	UNALLOCATED	-
00	001xx	UNALLOCATED	-
00	01xxx	UNALLOCATED	-
00	1xxxx	UNALLOCATED	-
!= 00		UNALLOCATED	-

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C4.1.5.4 Advanced SIMD scalar copy

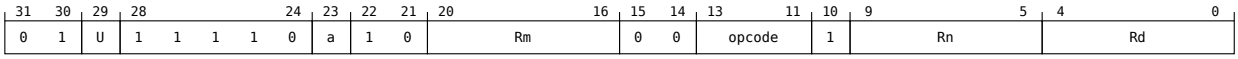
The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.



op	imm4	Instruction Details	Feature
0	0000	DUP (element)	FEAT_AdvSIMD
0	!= 0000	UNALLOCATED	-
1		UNALLOCATED	-

C4.1.5.5 Advanced SIMD scalar three same FP16

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

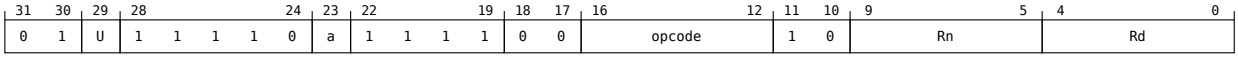


U	a	opcode	Instruction Details	Feature
		00x	UNALLOCATED	-
	0	010	UNALLOCATED	-
0		101	UNALLOCATED	-
0	0	011	FMULX	FEAT_AdvSIMD && FEAT_FP16
0	0	100	FCMEQ (register)	FEAT_AdvSIMD && FEAT_FP16
0	0	110	UNALLOCATED	-
0	0	111	FRECPS	FEAT_AdvSIMD && FEAT_FP16
0	1	x10	UNALLOCATED	-
0	1	011	UNALLOCATED	-
0	1	100	UNALLOCATED	-
0	1	111	FRSQRTS	FEAT_AdvSIMD && FEAT_FP16
1		011	UNALLOCATED	-
1		11x	UNALLOCATED	-
1	0	100	FCMGE (register)	FEAT_AdvSIMD && FEAT_FP16
1	0	101	FACGE	FEAT_AdvSIMD && FEAT_FP16
1	1	010	FABD	FEAT_AdvSIMD && FEAT_FP16
1	1	100	FCMGT (register)	FEAT_AdvSIMD && FEAT_FP16
1	1	101	FACGT	FEAT_AdvSIMD && FEAT_FP16

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C4.1.5.6 Advanced SIMD scalar two-register miscellaneous FP16

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

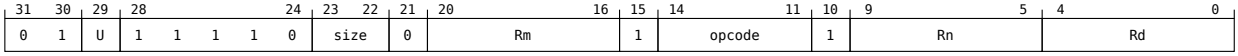


U	a	opcode	Instruction Details	Feature
		x0xxx	UNALLOCATED	-
		1100x	UNALLOCATED	-
	0	01xxx	UNALLOCATED	-
	1	010xx	UNALLOCATED	-
	1	11100	UNALLOCATED	-
0	0	11010	FCVTNS (vector)	FEAT_AdvSIMD && FEAT_FP16
0	0	11011	FCVTMS (vector)	FEAT_AdvSIMD && FEAT_FP16
0	0	11100	FCVTAS (vector)	FEAT_AdvSIMD && FEAT_FP16
0	0	11101	SCVTF (vector, integer)	FEAT_AdvSIMD && FEAT_FP16
0	0	1111x	UNALLOCATED	-
0	1	01100	FCMGT (zero)	FEAT_AdvSIMD && FEAT_FP16
0	1	01101	FCMEQ (zero)	FEAT_AdvSIMD && FEAT_FP16
0	1	01110	FCMLT (zero)	FEAT_AdvSIMD && FEAT_FP16
0	1	01111	UNALLOCATED	-
0	1	11010	FCVTPS (vector)	FEAT_AdvSIMD && FEAT_FP16
0	1	11011	FCVTZS (vector, integer)	FEAT_AdvSIMD && FEAT_FP16
0	1	11101	FRECPE	FEAT_AdvSIMD && FEAT_FP16
0	1	11110	UNALLOCATED	-
0	1	11111	FRECPX	FEAT_AdvSIMD && FEAT_FP16
1		1111x	UNALLOCATED	-
1	0	11010	FCVTNU (vector)	FEAT_AdvSIMD && FEAT_FP16
1	0	11011	FCVTMU (vector)	FEAT_AdvSIMD && FEAT_FP16
1	0	11100	FCVTAU (vector)	FEAT_AdvSIMD && FEAT_FP16
1	0	11101	UCVTF (vector, integer)	FEAT_AdvSIMD && FEAT_FP16
1	1	01100	FCMGE (zero)	FEAT_AdvSIMD && FEAT_FP16
1	1	01101	FCMLE (zero)	FEAT_AdvSIMD && FEAT_FP16
1	1	0111x	UNALLOCATED	-
1	1	11010	FCVTPU (vector)	FEAT_AdvSIMD && FEAT_FP16
1	1	11011	FCVTZU (vector, integer)	FEAT_AdvSIMD && FEAT_FP16
1	1	11101	FRSQRTE	FEAT_AdvSIMD && FEAT_FP16

C4.1.5.7 Advanced SIMD scalar three same extra

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

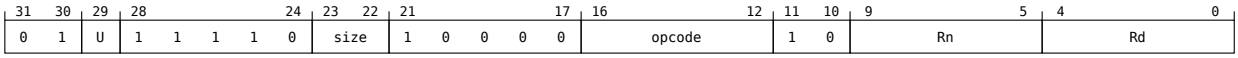
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential



U	opcode	Instruction Details	Feature
0		UNALLOCATED	-
1	0000	SQRDMLAH (vector)	FEAT_RDM
1	0001	SQRDMLSH (vector)	FEAT_RDM
1	001x	UNALLOCATED	-
1	01xx	UNALLOCATED	-
1	1xxx	UNALLOCATED	-

C4.1.5.8 Advanced SIMD scalar two-register miscellaneous

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.



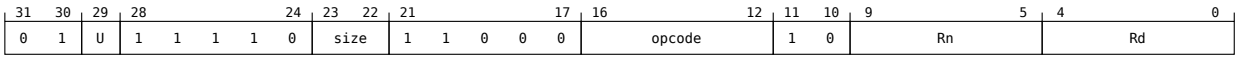
U	size	opcode	Instruction Details	Feature
		00xx0	UNALLOCATED	-
		00x01	UNALLOCATED	-
		1x000	UNALLOCATED	-
		10xx1	UNALLOCATED	-
		11001	UNALLOCATED	-
	0x	01xxx	UNALLOCATED	-
	0x	1111x	UNALLOCATED	-
	1x	01111	UNALLOCATED	-
	1x	11100	UNALLOCATED	-
	10	010x1	UNALLOCATED	-
	10	01000	UNALLOCATED	-
0		00011	SUQADD	FEAT_AdvSIMD
0		00111	SQABS	FEAT_AdvSIMD
0		10010	UNALLOCATED	-
0		10100	SQXTN, SQXTN2	FEAT_AdvSIMD
0		10110	UNALLOCATED	-
0	0x	11010	FCVTNS (vector)	FEAT_AdvSIMD
0	0x	11011	FCVTMS (vector)	FEAT_AdvSIMD
0	0x	11100	FCVTAS (vector)	FEAT_AdvSIMD
0	0x	11101	SCVTF (vector, integer)	FEAT_AdvSIMD
0	1x	01100	FCMGT (zero)	FEAT_AdvSIMD

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	size	opcode	Instruction Details	Feature
0	1x	01101	FCMEQ (zero)	FEAT_AdvSIMD
0	1x	01110	FCMLT (zero)	FEAT_AdvSIMD
0	1x	11010	FCVTPS (vector)	FEAT_AdvSIMD
0	1x	11011	FCVTZS (vector, integer)	FEAT_AdvSIMD
0	1x	11101	FRECPE	FEAT_AdvSIMD
0	1x	11110	UNALLOCATED	-
0	1x	11111	FRECPX	FEAT_AdvSIMD
0	10	01010	UNALLOCATED	-
0	11	01000	CMGT (zero)	FEAT_AdvSIMD
0	11	01001	CMEQ (zero)	FEAT_AdvSIMD
0	11	01010	CMLT (zero)	FEAT_AdvSIMD
0	11	01011	ABS	FEAT_AdvSIMD
1		00011	USQADD	FEAT_AdvSIMD
1		00111	SQNEG	FEAT_AdvSIMD
1		10010	SQXTUN, SQXTUN2	FEAT_AdvSIMD
1		10100	UQXTN, UQXTN2	FEAT_AdvSIMD
1	0x	11010	FCVTNU (vector)	FEAT_AdvSIMD
1	0x	11011	FCVTMU (vector)	FEAT_AdvSIMD
1	0x	11100	FCVTAU (vector)	FEAT_AdvSIMD
1	0x	11101	UCVTF (vector, integer)	FEAT_AdvSIMD
1	00	10110	UNALLOCATED	-
1	01	10110	FCVTXN, FCVTXN2	FEAT_AdvSIMD
1	1x	01x10	UNALLOCATED	-
1	1x	01100	FCMGE (zero)	FEAT_AdvSIMD
1	1x	01101	FCMLE (zero)	FEAT_AdvSIMD
1	1x	1x110	UNALLOCATED	-
1	1x	11010	FCVTPU (vector)	FEAT_AdvSIMD
1	1x	11011	FCVTZU (vector, integer)	FEAT_AdvSIMD
1	1x	11101	FRSQRTE	FEAT_AdvSIMD
1	1x	11111	UNALLOCATED	-
1	11	01000	CMGE (zero)	FEAT_AdvSIMD
1	11	01001	CMLE (zero)	FEAT_AdvSIMD
1	11	01011	NEG (vector)	FEAT_AdvSIMD

C4.1.5.9 Advanced SIMD scalar pairwise

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

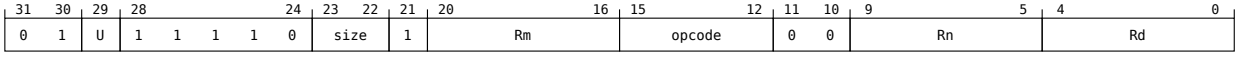


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	size	opcode	Instruction Details	Feature
		x0xxx	UNALLOCATED	-
		01110	UNALLOCATED	-
		111xx	UNALLOCATED	-
	10	01101	UNALLOCATED	-
0	x1	011x1	UNALLOCATED	-
0	x1	01100	UNALLOCATED	-
0	00	01100	FMAXNMP (scalar) — Half-precision	FEAT_AdvSIMD && FEAT_FP16
0	00	01101	FADDP (scalar) — Half-precision	FEAT_AdvSIMD && FEAT_FP16
0	00	01111	FMAXP (scalar) — Half-precision	FEAT_AdvSIMD && FEAT_FP16
0	10	01100	FMINNMP (scalar) — Half-precision	FEAT_AdvSIMD && FEAT_FP16
0	10	01111	FMINP (scalar) — Half-precision	FEAT_AdvSIMD && FEAT_FP16
0	11	x100x	UNALLOCATED	-
0	11	x1010	UNALLOCATED	-
0	11	01011	UNALLOCATED	-
0	11	11011	ADDP (scalar)	FEAT_AdvSIMD
0	!= 11	x10xx	UNALLOCATED	-
1		x10xx	UNALLOCATED	-
1	0x	01100	FMAXNMP (scalar) — Single-precision and double-precision	FEAT_AdvSIMD
1	0x	01101	FADDP (scalar) — Single-precision and double-precision	FEAT_AdvSIMD
1	0x	01111	FMAXP (scalar) — Single-precision and double-precision	FEAT_AdvSIMD
1	1x	01100	FMINNMP (scalar) — Single-precision and double-precision	FEAT_AdvSIMD
1	1x	01111	FMINP (scalar) — Single-precision and double-precision	FEAT_AdvSIMD
1	11	01101	UNALLOCATED	-

C4.1.5.10 Advanced SIMD scalar three different

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.



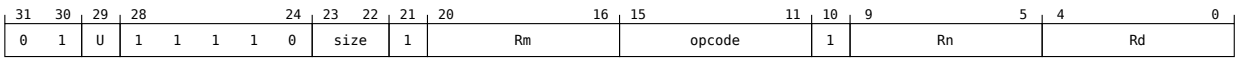
U	opcode	Instruction Details	Feature
0	0xxx	UNALLOCATED	-
0	1xx0	UNALLOCATED	-
0	1001	SQDMLAL, SQDMLAL2 (vector)	FEAT_AdvSIMD
0	1011	SQDMLSL, SQDMLSL2 (vector)	FEAT_AdvSIMD

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	opcode	Instruction Details	Feature
0	1101	SQDMULL, SQDMULL2 (vector)	FEAT_AdvSIMD
0	1111	UNALLOCATED	-
1		UNALLOCATED	-

C4.1.5.11 Advanced SIMD scalar three same

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.



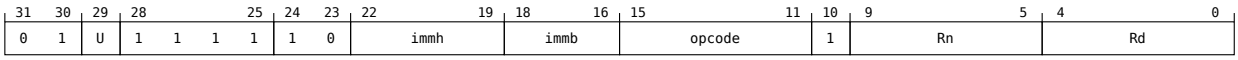
U	size	opcode	Instruction Details	Feature
		x0011	UNALLOCATED	-
		011x1	UNALLOCATED	-
		11001	UNALLOCATED	-
		11110	UNALLOCATED	-
	x0	0x1x0	UNALLOCATED	-
	x1	00100	UNALLOCATED	-
	x1	011x0	UNALLOCATED	-
	0x	xx0x0	UNALLOCATED	-
	0x	10x01	UNALLOCATED	-
	01	00110	UNALLOCATED	-
	1x	10010	UNALLOCATED	-
	1x	11000	UNALLOCATED	-
	10	0x0x0	UNALLOCATED	-
	10	10x0x	UNALLOCATED	-
	11	00000	UNALLOCATED	-
	11	00010	UNALLOCATED	-
	11	10101	UNALLOCATED	-
	!= 10	10100	UNALLOCATED	-
	!= 11	00111	UNALLOCATED	-
0		00001	SQADD	FEAT_AdvSIMD
0		00101	SQSUB	FEAT_AdvSIMD
0		01001	SQSHL (register)	FEAT_AdvSIMD
0		01011	SQRSHL	FEAT_AdvSIMD
0		10110	SQDMULH (vector)	FEAT_AdvSIMD
0		10111	UNALLOCATED	-
0		11101	UNALLOCATED	-
0	0x	11011	FMULX	FEAT_AdvSIMD

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	size	opcode	Instruction Details	Feature
0	0x	11100	FCMEQ (register)	FEAT_AdvSIMD
0	0x	11111	FRECPS	FEAT_AdvSIMD
0	1x	11010	UNALLOCATED	-
0	1x	11011	UNALLOCATED	-
0	1x	11100	UNALLOCATED	-
0	1x	11111	FRSQRTS	FEAT_AdvSIMD
0	11	00110	CMGT (register)	FEAT_AdvSIMD
0	11	00111	CMGE (register)	FEAT_AdvSIMD
0	11	01000	SSHL	FEAT_AdvSIMD
0	11	01010	SRSHL	FEAT_AdvSIMD
0	11	10000	ADD (vector)	FEAT_AdvSIMD
0	11	10001	CMTST	FEAT_AdvSIMD
1		00001	UQADD	FEAT_AdvSIMD
1		00101	UQSUB	FEAT_AdvSIMD
1		01001	UQSHL (register)	FEAT_AdvSIMD
1		01011	UQRSHL	FEAT_AdvSIMD
1		1x111	UNALLOCATED	-
1		10110	SQRDMULH (vector)	FEAT_AdvSIMD
1		11011	UNALLOCATED	-
1	0x	11100	FCMGE (register)	FEAT_AdvSIMD
1	0x	11101	FACGE	FEAT_AdvSIMD
1	1x	11010	FABD	FEAT_AdvSIMD
1	1x	11100	FCMGT (register)	FEAT_AdvSIMD
1	1x	11101	FACGT	FEAT_AdvSIMD
1	11	00110	CMHI (register)	FEAT_AdvSIMD
1	11	00111	CMHS (register)	FEAT_AdvSIMD
1	11	01000	USHL	FEAT_AdvSIMD
1	11	01010	URSHL	FEAT_AdvSIMD
1	11	10000	SUB (vector)	FEAT_AdvSIMD
1	11	10001	CMEQ (register)	FEAT_AdvSIMD

C4.1.5.12 Advanced SIMD scalar shift by immediate

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.



Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

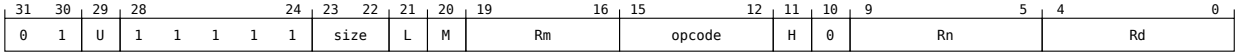
U	immh	opcode	Instruction Details	Feature
		0xxx1	UNALLOCATED	-
		101xx	UNALLOCATED	-
		11101	UNALLOCATED	-
	0xxx	x10x0	UNALLOCATED	-
	0xxx	00xx0	UNALLOCATED	-
	0xxx	110x1	UNALLOCATED	-
	0000	x1110	UNALLOCATED	-
	0000	1001x	UNALLOCATED	-
	0000	11100	UNALLOCATED	-
	0000	11111	UNALLOCATED	-
	1xxx	110xx	UNALLOCATED	-
	!= 0000	11110	UNALLOCATED	-
0		01100	UNALLOCATED	-
0		1000x	UNALLOCATED	-
0	1xxx	00000	SSHR	FEAT_AdvSIMD
0	1xxx	00010	SSRA	FEAT_AdvSIMD
0	1xxx	00100	SRSHR	FEAT_AdvSIMD
0	1xxx	00110	SRSRA	FEAT_AdvSIMD
0	1xxx	01000	UNALLOCATED	-
0	1xxx	01010	SHL	FEAT_AdvSIMD
0	!= 0000	01110	SQSHL (immediate)	FEAT_AdvSIMD
0	!= 0000	10010	SQSHRN, SQSHRN2	FEAT_AdvSIMD
0	!= 0000	10011	SQRSHRN, SQRSHRN2	FEAT_AdvSIMD
0	!= 0000	11100	SCVTF (vector, fixed-point)	FEAT_AdvSIMD
0	!= 0000	11111	FCVTZS (vector, fixed-point)	FEAT_AdvSIMD
1	0000	01100	UNALLOCATED	-
1	0000	1000x	UNALLOCATED	-
1	1xxx	00000	USHR	FEAT_AdvSIMD
1	1xxx	00010	USRA	FEAT_AdvSIMD
1	1xxx	00100	URSHR	FEAT_AdvSIMD
1	1xxx	00110	URSRA	FEAT_AdvSIMD
1	1xxx	01000	SRI	FEAT_AdvSIMD
1	1xxx	01010	SLI	FEAT_AdvSIMD
1	!= 0000	01100	SQSHLU	FEAT_AdvSIMD
1	!= 0000	01110	UQSHL (immediate)	FEAT_AdvSIMD
1	!= 0000	10000	SQSHRUN, SQSHRUN2	FEAT_AdvSIMD
1	!= 0000	10001	SQRSHRUN, SQRSHRUN2	FEAT_AdvSIMD
1	!= 0000	10010	UQSHRN, UQSHRN2	FEAT_AdvSIMD
1	!= 0000	10011	UQRSHRN, UQRSHRN2	FEAT_AdvSIMD
1	!= 0000	11100	UCVTF (vector, fixed-point)	FEAT_AdvSIMD

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	immh	opcode	Instruction Details	Feature
1	!= 0000	11111	FCVTZU (vector, fixed-point)	FEAT_AdvSIMD

C4.1.5.13 Advanced SIMD scalar x indexed element

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

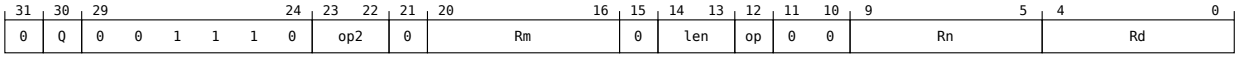


U	size	opcode	Instruction Details	Feature
	01	1001	UNALLOCATED	-
0		0xx0	UNALLOCATED	-
0		0011	SQDMLAL, SQDMLAL2 (by element)	FEAT_AdvSIMD
0		0111	SQDMLSL, SQDMLSL2 (by element)	FEAT_AdvSIMD
0		10x0	UNALLOCATED	-
0		1011	SQDMULL, SQDMULL2 (by element)	FEAT_AdvSIMD
0		1100	SQDMULH (by element)	FEAT_AdvSIMD
0		1101	SQRDMULH (by element)	FEAT_AdvSIMD
0		111x	UNALLOCATED	-
0	00	0001	FMLA (by element) — Scalar, half-precision	FEAT_AdvSIMD && FEAT_FP16
0	00	0101	FMLS (by element) — Scalar, half-precision	FEAT_AdvSIMD && FEAT_FP16
0	00	1001	FMUL (by element) — Scalar, half-precision	FEAT_AdvSIMD && FEAT_FP16
0	01	0x01	UNALLOCATED	-
0	1x	0001	FMLA (by element) — Scalar, single-precision and double-precision	FEAT_AdvSIMD
0	1x	0101	FMLS (by element) — Scalar, single-precision and double-precision	FEAT_AdvSIMD
0	1x	1001	FMUL (by element) — Scalar, single-precision and double-precision	FEAT_AdvSIMD
1		0xxx	UNALLOCATED	-
1		1xx0	UNALLOCATED	-
1		1011	UNALLOCATED	-
1		1101	SQRDMLAH (by element)	FEAT_RDM
1		1111	SQRDMLSH (by element)	FEAT_RDM
1	00	1001	FMULX (by element) — Scalar, half-precision	FEAT_AdvSIMD && FEAT_FP16
1	1x	1001	FMULX (by element) — Scalar, single-precision and double-precision	FEAT_AdvSIMD

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C4.1.5.14 Advanced SIMD table lookup

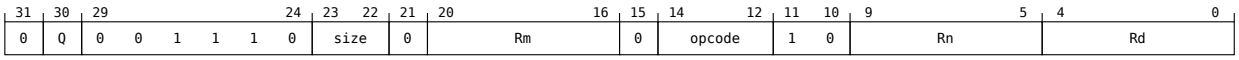
The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.



op2	len	op	Instruction Details	Feature
00	00	0	TBL — Single register table	FEAT_AdvSIMD
00	00	1	TBX — Single register table	FEAT_AdvSIMD
00	01	0	TBL — Two register table	FEAT_AdvSIMD
00	01	1	TBX — Two register table	FEAT_AdvSIMD
00	10	0	TBL — Three register table	FEAT_AdvSIMD
00	10	1	TBX — Three register table	FEAT_AdvSIMD
00	11	0	TBL — Four register table	FEAT_AdvSIMD
00	11	1	TBX — Four register table	FEAT_AdvSIMD
!= 00			UNALLOCATED	-

C4.1.5.15 Advanced SIMD permute

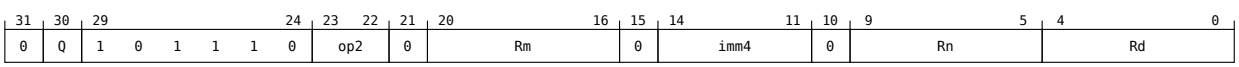
The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.



opcode	Instruction Details	Feature
x00	UNALLOCATED	-
001	UZP1	FEAT_AdvSIMD
010	TRN1	FEAT_AdvSIMD
011	ZIP1	FEAT_AdvSIMD
101	UZP2	FEAT_AdvSIMD
110	TRN2	FEAT_AdvSIMD
111	ZIP2	FEAT_AdvSIMD

C4.1.5.16 Advanced SIMD extract

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

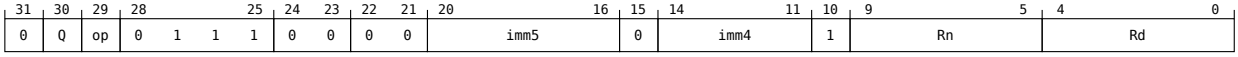


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op2	Instruction Details	Feature
00	EXT	FEAT_AdvSIMD
!= 00	UNALLOCATED	-

C4.1.5.17 Advanced SIMD copy

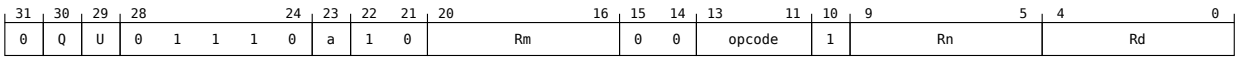
The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.



Q	op	imm5	imm4	Instruction Details	Feature
	0		0000	DUP (element)	FEAT_AdvSIMD
	0		0001	DUP (general)	FEAT_AdvSIMD
	0		01x0	UNALLOCATED	-
	0		1xxx	UNALLOCATED	-
0	0		001x	UNALLOCATED	-
0	0		0101	SMOV — 32-bit	FEAT_AdvSIMD
0	0		0111	UMOV — 32-bit	FEAT_AdvSIMD
0	1			UNALLOCATED	-
1	0		0010	UNALLOCATED	-
1	0		0011	INS (general)	FEAT_AdvSIMD
1	0		0101	SMOV — 64-bit	FEAT_AdvSIMD
1	0	x0xxx	0111	UNALLOCATED	-
1	0	x1000	0111	UMOV — 64-bit	FEAT_AdvSIMD
1	0	x1001	0111	UNALLOCATED	-
1	0	x101x	0111	UNALLOCATED	-
1	0	x11xx	0111	UNALLOCATED	-
1	1			INS (element)	FEAT_AdvSIMD

C4.1.5.18 Advanced SIMD three same (FP16)

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.



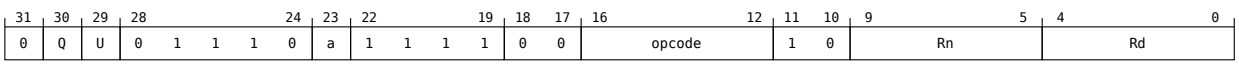
U	a	opcode	Instruction Details	Feature
0	0	000	FMAXNM (vector)	FEAT_AdvSIMD && FEAT_FP16
0	0	001	FMLA (vector)	FEAT_AdvSIMD && FEAT_FP16

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	a	opcode	Instruction Details	Feature
0	0	010	FADD (vector)	FEAT_AdvSIMD && FEAT_FP16
0	0	011	FMULX	FEAT_AdvSIMD && FEAT_FP16
0	0	100	FCMEQ (register)	FEAT_AdvSIMD && FEAT_FP16
0	0	101	UNALLOCATED	-
0	0	110	FMAX (vector)	FEAT_AdvSIMD && FEAT_FP16
0	0	111	FRECPS	FEAT_AdvSIMD && FEAT_FP16
0	1	000	FMINNM (vector)	FEAT_AdvSIMD && FEAT_FP16
0	1	001	FMLS (vector)	FEAT_AdvSIMD && FEAT_FP16
0	1	010	FSUB (vector)	FEAT_AdvSIMD && FEAT_FP16
0	1	011	UNALLOCATED	-
0	1	10x	UNALLOCATED	-
0	1	110	FMIN (vector)	FEAT_AdvSIMD && FEAT_FP16
0	1	111	FRSQRTS	FEAT_AdvSIMD && FEAT_FP16
1		001	UNALLOCATED	-
1	0	000	FMAXNMP (vector)	FEAT_AdvSIMD && FEAT_FP16
1	0	010	FADDP (vector)	FEAT_AdvSIMD && FEAT_FP16
1	0	011	FMUL (vector)	FEAT_AdvSIMD && FEAT_FP16
1	0	100	FCMGE (register)	FEAT_AdvSIMD && FEAT_FP16
1	0	101	FACGE	FEAT_AdvSIMD && FEAT_FP16
1	0	110	FMAXP (vector)	FEAT_AdvSIMD && FEAT_FP16
1	0	111	FDIV (vector)	FEAT_AdvSIMD && FEAT_FP16
1	1	x11	UNALLOCATED	-
1	1	000	FMINNMP (vector)	FEAT_AdvSIMD && FEAT_FP16
1	1	010	FABD	FEAT_AdvSIMD && FEAT_FP16
1	1	100	FCMGT (register)	FEAT_AdvSIMD && FEAT_FP16
1	1	101	FACGT	FEAT_AdvSIMD && FEAT_FP16
1	1	110	FMINP (vector)	FEAT_AdvSIMD && FEAT_FP16

C4.1.5.19 Advanced SIMD two-register miscellaneous (FP16)

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.



U	a	opcode	Instruction Details	Feature
		x0xxx	UNALLOCATED	-
	0	01xxx	UNALLOCATED	-
	0	1111x	UNALLOCATED	-

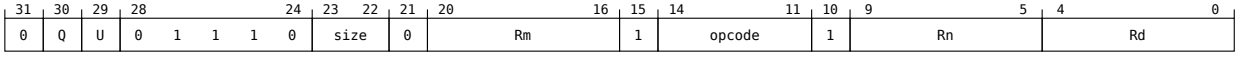
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	a	opcode	Instruction Details	Feature
	1	010xx	UNALLOCATED	-
	1	11100	UNALLOCATED	-
0	0	11000	FRINTN (vector)	FEAT_AdvSIMD && FEAT_FP16
0	0	11001	FRINTM (vector)	FEAT_AdvSIMD && FEAT_FP16
0	0	11010	FCVTNS (vector)	FEAT_AdvSIMD && FEAT_FP16
0	0	11011	FCVTMS (vector)	FEAT_AdvSIMD && FEAT_FP16
0	0	11100	FCVTAS (vector)	FEAT_AdvSIMD && FEAT_FP16
0	0	11101	SCVTF (vector, integer)	FEAT_AdvSIMD && FEAT_FP16
0	1	01100	FCMGT (zero)	FEAT_AdvSIMD && FEAT_FP16
0	1	01101	FCMEQ (zero)	FEAT_AdvSIMD && FEAT_FP16
0	1	01110	FCMLT (zero)	FEAT_AdvSIMD && FEAT_FP16
0	1	01111	FABS (vector)	FEAT_AdvSIMD && FEAT_FP16
0	1	11000	FRINTP (vector)	FEAT_AdvSIMD && FEAT_FP16
0	1	11001	FRINTZ (vector)	FEAT_AdvSIMD && FEAT_FP16
0	1	11010	FCVTPS (vector)	FEAT_AdvSIMD && FEAT_FP16
0	1	11011	FCVTZS (vector, integer)	FEAT_AdvSIMD && FEAT_FP16
0	1	11101	FRECPE	FEAT_AdvSIMD && FEAT_FP16
0	1	1111x	UNALLOCATED	-
1	0	11000	FRINTA (vector)	FEAT_AdvSIMD && FEAT_FP16
1	0	11001	FRINTX (vector)	FEAT_AdvSIMD && FEAT_FP16
1	0	11010	FCVTNU (vector)	FEAT_AdvSIMD && FEAT_FP16
1	0	11011	FCVTMU (vector)	FEAT_AdvSIMD && FEAT_FP16
1	0	11100	FCVTAU (vector)	FEAT_AdvSIMD && FEAT_FP16
1	0	11101	UCVTF (vector, integer)	FEAT_AdvSIMD && FEAT_FP16
1	1	x1110	UNALLOCATED	-
1	1	01100	FCMGE (zero)	FEAT_AdvSIMD && FEAT_FP16
1	1	01101	FCMLE (zero)	FEAT_AdvSIMD && FEAT_FP16
1	1	01111	FNEG (vector)	FEAT_AdvSIMD && FEAT_FP16
1	1	11000	UNALLOCATED	-
1	1	11001	FRINTI (vector)	FEAT_AdvSIMD && FEAT_FP16
1	1	11010	FCVTPU (vector)	FEAT_AdvSIMD && FEAT_FP16
1	1	11011	FCVTZU (vector, integer)	FEAT_AdvSIMD && FEAT_FP16
1	1	11101	FRSQRTE	FEAT_AdvSIMD && FEAT_FP16
1	1	11111	FSQRT (vector)	FEAT_AdvSIMD && FEAT_FP16

C4.1.5.20 Advanced SIMD three-register extension

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

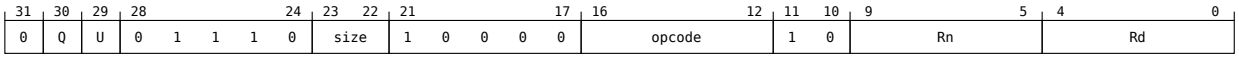
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential



U	opcode	Instruction Details	Feature
	0011	UNALLOCATED	-
	011x	UNALLOCATED	-
0	0x0x	UNALLOCATED	-
0	0010	SDOT (vector)	FEAT_DotProd
0	1xxx	UNALLOCATED	-
1	0000	SQRDMLAH (vector)	FEAT_RDM
1	0001	SQRDMLSH (vector)	FEAT_RDM
1	0010	UDOT (vector)	FEAT_DotProd
1	010x	UNALLOCATED	-
1	10xx	FCMLA	FEAT_FCMA
1	11x0	FCADD	FEAT_FCMA
1	11x1	UNALLOCATED	-

C4.1.5.21 Advanced SIMD two-register miscellaneous

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.



U	size	opcode	Instruction Details	Feature
		1000x	UNALLOCATED	-
		10101	UNALLOCATED	-
	0x	011xx	UNALLOCATED	-
	1x	1011x	UNALLOCATED	-
0		00000	REV64	FEAT_AdvSIMD
0		00001	REV16 (vector)	FEAT_AdvSIMD
0		00010	SADDLP	FEAT_AdvSIMD
0		00011	SUQADD	FEAT_AdvSIMD
0		00100	CLS (vector)	FEAT_AdvSIMD
0		00101	CNT	FEAT_AdvSIMD
0		00110	SADALP	FEAT_AdvSIMD
0		00111	SQABS	FEAT_AdvSIMD
0		01000	CMGT (zero)	FEAT_AdvSIMD
0		01001	CMEQ (zero)	FEAT_AdvSIMD
0		01010	CMLT (zero)	FEAT_AdvSIMD

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

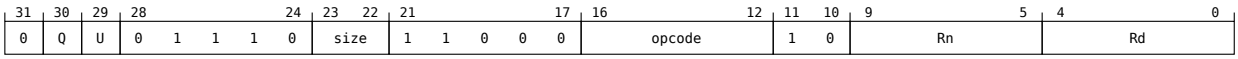
U	size	opcode	Instruction Details	Feature
0		01011	ABS	FEAT_AdvSIMD
0		10010	XTN, XTN2	FEAT_AdvSIMD
0		10011	UNALLOCATED	-
0		10100	SQXTN, SQXTN2	FEAT_AdvSIMD
0		1111x	UNALLOCATED	-
0	0x	10110	FCVTN, FCVTN2 (double to single-precision, single to half-precision)	FEAT_AdvSIMD
0	0x	10111	FCVTL, FCVTL2	FEAT_AdvSIMD
0	0x	11000	FRINTN (vector)	FEAT_AdvSIMD
0	0x	11001	FRINTM (vector)	FEAT_AdvSIMD
0	0x	11010	FCVTNS (vector)	FEAT_AdvSIMD
0	0x	11011	FCVTMS (vector)	FEAT_AdvSIMD
0	0x	11100	FCVTAS (vector)	FEAT_AdvSIMD
0	0x	11101	SCVTF (vector, integer)	FEAT_AdvSIMD
0	1x	01100	FCMGT (zero)	FEAT_AdvSIMD
0	1x	01101	FCMEQ (zero)	FEAT_AdvSIMD
0	1x	01110	FCMLT (zero)	FEAT_AdvSIMD
0	1x	01111	FABS (vector)	FEAT_AdvSIMD
0	1x	11000	FRINTP (vector)	FEAT_AdvSIMD
0	1x	11001	FRINTZ (vector)	FEAT_AdvSIMD
0	1x	11010	FCVTPS (vector)	FEAT_AdvSIMD
0	1x	11011	FCVTZS (vector, integer)	FEAT_AdvSIMD
0	1x	11100	URECPE	FEAT_AdvSIMD
0	1x	11101	FRECPE	FEAT_AdvSIMD
1		00000	REV32 (vector)	FEAT_AdvSIMD
1		00010	UADDLP	FEAT_AdvSIMD
1		00011	USQADD	FEAT_AdvSIMD
1		00100	CLZ (vector)	FEAT_AdvSIMD
1		00110	UADALP	FEAT_AdvSIMD
1		00111	SQNEG	FEAT_AdvSIMD
1		01000	CMGE (zero)	FEAT_AdvSIMD
1		01001	CMLE (zero)	FEAT_AdvSIMD
1		01010	UNALLOCATED	-
1		01011	NEG (vector)	FEAT_AdvSIMD
1		10010	SQXTUN, SQXTUN2	FEAT_AdvSIMD
1		10011	SHLL, SHLL2	FEAT_AdvSIMD
1		10100	UQXTN, UQXTN2	FEAT_AdvSIMD
1		11110	UNALLOCATED	-
1	0x	00001	UNALLOCATED	-
1	0x	1x111	UNALLOCATED	-
1	0x	11000	FRINTA (vector)	FEAT_AdvSIMD

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	size	opcode	Instruction Details	Feature
1	0x	11001	FRINTX (vector)	FEAT_AdvSIMD
1	0x	11010	FCVTNU (vector)	FEAT_AdvSIMD
1	0x	11011	FCVTMU (vector)	FEAT_AdvSIMD
1	0x	11100	FCVTAU (vector)	FEAT_AdvSIMD
1	0x	11101	UCVTF (vector, integer)	FEAT_AdvSIMD
1	00	00101	NOT	FEAT_AdvSIMD
1	00	10110	UNALLOCATED	-
1	01	00101	RBIT (vector)	FEAT_AdvSIMD
1	01	10110	FCVTXN, FCVTXN2	FEAT_AdvSIMD
1	1x	00x01	UNALLOCATED	-
1	1x	01100	FCMGE (zero)	FEAT_AdvSIMD
1	1x	01101	FCMLE (zero)	FEAT_AdvSIMD
1	1x	01110	UNALLOCATED	-
1	1x	01111	FNEG (vector)	FEAT_AdvSIMD
1	1x	11000	UNALLOCATED	-
1	1x	11001	FRINTI (vector)	FEAT_AdvSIMD
1	1x	11010	FCVTPU (vector)	FEAT_AdvSIMD
1	1x	11011	FCVTZU (vector, integer)	FEAT_AdvSIMD
1	1x	11100	URSQRTE	FEAT_AdvSIMD
1	1x	11101	FRSQRTE	FEAT_AdvSIMD
1	1x	11111	FSQRT (vector)	FEAT_AdvSIMD

C4.1.5.22 Advanced SIMD across lanes

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.



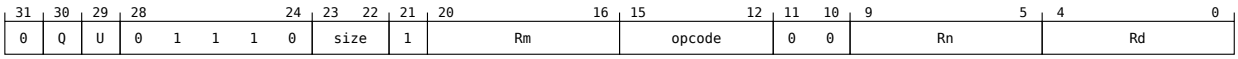
Q	U	size	opcode	Instruction Details	Feature
			x100x	UNALLOCATED	-
			000x0	UNALLOCATED	-
			00001	UNALLOCATED	-
			01011	UNALLOCATED	-
			100xx	UNALLOCATED	-
		x0	001xx	UNALLOCATED	-
		x0	01101	UNALLOCATED	-
		x0	101xx	UNALLOCATED	-
		x0	111xx	UNALLOCATED	-

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q	U	size	opcode	Instruction Details	Feature
		x1	xx1xx	UNALLOCATED	-
	0		00011	SADDLV	FEAT_AdvSIMD
	0		01010	SMAXV	FEAT_AdvSIMD
	0		11010	SMINV	FEAT_AdvSIMD
	0		11011	ADDV	FEAT_AdvSIMD
	0	x0	01110	UNALLOCATED	-
	0	00	01100	FMAXNMV — Half-precision	FEAT_AdvSIMD && FEAT_FP16
	0	00	01111	FMAXV — Half-precision	FEAT_AdvSIMD && FEAT_FP16
	0	10	01100	FMINNMV — Half-precision	FEAT_AdvSIMD && FEAT_FP16
	0	10	01111	FMINV — Half-precision	FEAT_AdvSIMD && FEAT_FP16
	1		00011	UADDLV	FEAT_AdvSIMD
	1		01010	UMAXV	FEAT_AdvSIMD
	1		11010	UMINV	FEAT_AdvSIMD
	1		11011	UNALLOCATED	-
0	1	x0	011x0	UNALLOCATED	-
0	1	x0	01111	UNALLOCATED	-
1	1	x0	01110	UNALLOCATED	-
1	1	00	01100	FMAXNMV — Single-precision	FEAT_AdvSIMD
1	1	00	01111	FMAXV — Single-precision	FEAT_AdvSIMD
1	1	10	01100	FMINNMV — Single-precision	FEAT_AdvSIMD
1	1	10	01111	FMINV — Single-precision	FEAT_AdvSIMD

C4.1.5.23 Advanced SIMD three different

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.



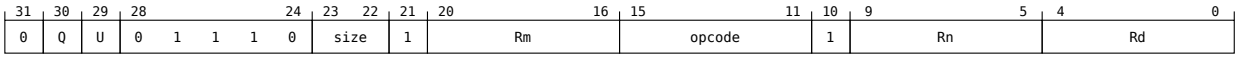
U	opcode	Instruction Details	Feature
0	0000	SADDL, SADDL2	FEAT_AdvSIMD
0	0001	SADDW, SADDW2	FEAT_AdvSIMD
0	0010	SSUBL, SSUBL2	FEAT_AdvSIMD
0	0011	SSUBW, SSUBW2	FEAT_AdvSIMD
0	0100	ADDHN, ADDHN2	FEAT_AdvSIMD
0	0101	SABAL, SABAL2	FEAT_AdvSIMD
0	0110	SUBHN, SUBHN2	FEAT_AdvSIMD
0	0111	SABDL, SABDL2	FEAT_AdvSIMD
0	1000	SMLAL, SMLAL2 (vector)	FEAT_AdvSIMD

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	opcode	Instruction Details	Feature
0	1001	SQDMLAL, SQDMLAL2 (vector)	FEAT_AdvSIMD
0	1010	SMLSL, SMLSL2 (vector)	FEAT_AdvSIMD
0	1011	SQDMLSL, SQDMLSL2 (vector)	FEAT_AdvSIMD
0	1100	SMULL, SMULL2 (vector)	FEAT_AdvSIMD
0	1101	SQDMULL, SQDMULL2 (vector)	FEAT_AdvSIMD
0	1110	PMULL, PMULL2	FEAT_AdvSIMD
0	1111	UNALLOCATED	-
1	0000	UADDL, UADDL2	FEAT_AdvSIMD
1	0001	UADDW, UADDW2	FEAT_AdvSIMD
1	0010	USUBL, USUBL2	FEAT_AdvSIMD
1	0011	USUBW, USUBW2	FEAT_AdvSIMD
1	0100	RADDHN, RADDHN2	FEAT_AdvSIMD
1	0101	UABAL, UABAL2	FEAT_AdvSIMD
1	0110	RSUBHN, RSUBHN2	FEAT_AdvSIMD
1	0111	UABDL, UABDL2	FEAT_AdvSIMD
1	1xx1	UNALLOCATED	-
1	1000	UMLAL, UMLAL2 (vector)	FEAT_AdvSIMD
1	1010	UMLSL, UMLSL2 (vector)	FEAT_AdvSIMD
1	1100	UMULL, UMULL2 (vector)	FEAT_AdvSIMD
1	1110	UNALLOCATED	-

C4.1.5.24 Advanced SIMD three same

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.



U	size	opcode	Instruction Details	Feature
	1x	11011	UNALLOCATED	-
0		00000	SHADD	FEAT_AdvSIMD
0		00001	SQADD	FEAT_AdvSIMD
0		00010	SRHADD	FEAT_AdvSIMD
0		00100	SHSUB	FEAT_AdvSIMD
0		00101	SQSUB	FEAT_AdvSIMD
0		00110	CMGT (register)	FEAT_AdvSIMD
0		00111	CMGE (register)	FEAT_AdvSIMD
0		01000	SSHL	FEAT_AdvSIMD
0		01001	SQSHL (register)	FEAT_AdvSIMD

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	size	opcode	Instruction Details	Feature
0		01010	SRSHL	FEAT_AdvSIMD
0		01011	SQRSHL	FEAT_AdvSIMD
0		01100	SMAX	FEAT_AdvSIMD
0		01101	SMIN	FEAT_AdvSIMD
0		01110	SABD	FEAT_AdvSIMD
0		01111	SABA	FEAT_AdvSIMD
0		10000	ADD (vector)	FEAT_AdvSIMD
0		10001	CMTST	FEAT_AdvSIMD
0		10010	MLA (vector)	FEAT_AdvSIMD
0		10011	MUL (vector)	FEAT_AdvSIMD
0		10100	SMAXP	FEAT_AdvSIMD
0		10101	SMINP	FEAT_AdvSIMD
0		10110	SQDMULH (vector)	FEAT_AdvSIMD
0		10111	ADDP (vector)	FEAT_AdvSIMD
0	x1	11101	UNALLOCATED	-
0	0x	11000	FMAXNM (vector)	FEAT_AdvSIMD
0	0x	11001	FMLA (vector)	FEAT_AdvSIMD
0	0x	11010	FADD (vector)	FEAT_AdvSIMD
0	0x	11011	FMULX	FEAT_AdvSIMD
0	0x	11100	FCMEQ (register)	FEAT_AdvSIMD
0	0x	11110	FMAX (vector)	FEAT_AdvSIMD
0	0x	11111	FRECPS	FEAT_AdvSIMD
0	00	00011	AND (vector)	FEAT_AdvSIMD
0	00	11101	FMLAL, FMLAL2 (vector) — FMLAL	FEAT_FHM
0	01	00011	BIC (vector, register)	FEAT_AdvSIMD
0	1x	11000	FMINNM (vector)	FEAT_AdvSIMD
0	1x	11001	FMLS (vector)	FEAT_AdvSIMD
0	1x	11010	FSUB (vector)	FEAT_AdvSIMD
0	1x	11100	UNALLOCATED	-
0	1x	11110	FMIN (vector)	FEAT_AdvSIMD
0	1x	11111	FRSQRTS	FEAT_AdvSIMD
0	10	00011	ORR (vector, register)	FEAT_AdvSIMD
0	10	11101	FMLSL, FMLSL2 (vector) — FMLSL	FEAT_FHM
0	11	00011	ORN (vector)	FEAT_AdvSIMD
1		00000	UHADD	FEAT_AdvSIMD
1		00001	UQADD	FEAT_AdvSIMD
1		00010	URHADD	FEAT_AdvSIMD
1		00100	UHSUB	FEAT_AdvSIMD
1		00101	UQSUB	FEAT_AdvSIMD
1		00110	CMHI (register)	FEAT_AdvSIMD

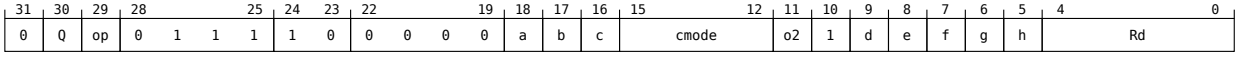
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	size	opcode	Instruction Details	Feature
1		00111	CMHS (register)	FEAT_AdvSIMD
1		01000	USHL	FEAT_AdvSIMD
1		01001	UQSHL (register)	FEAT_AdvSIMD
1		01010	URSHL	FEAT_AdvSIMD
1		01011	UQRSHL	FEAT_AdvSIMD
1		01100	UMAX	FEAT_AdvSIMD
1		01101	UMIN	FEAT_AdvSIMD
1		01110	UABD	FEAT_AdvSIMD
1		01111	UABA	FEAT_AdvSIMD
1		10000	SUB (vector)	FEAT_AdvSIMD
1		10001	CMEQ (register)	FEAT_AdvSIMD
1		10010	MLS (vector)	FEAT_AdvSIMD
1		10011	PMUL	FEAT_AdvSIMD
1		10100	UMAXP	FEAT_AdvSIMD
1		10101	UMINP	FEAT_AdvSIMD
1		10110	SQRDMULH (vector)	FEAT_AdvSIMD
1	x1	11001	UNALLOCATED	-
1	0x	10111	UNALLOCATED	-
1	0x	11000	FMAXNMP (vector)	FEAT_AdvSIMD
1	0x	11010	FADDP (vector)	FEAT_AdvSIMD
1	0x	11011	FMUL (vector)	FEAT_AdvSIMD
1	0x	11100	FCMGE (register)	FEAT_AdvSIMD
1	0x	11101	FACGE	FEAT_AdvSIMD
1	0x	11110	FMAXP (vector)	FEAT_AdvSIMD
1	0x	11111	FDIV (vector)	FEAT_AdvSIMD
1	00	00011	EOR (vector)	FEAT_AdvSIMD
1	00	11001	FMLAL, FMLAL2 (vector) — FMLAL2	FEAT_FHM
1	01	00011	BSL	FEAT_AdvSIMD
1	1x	1x111	UNALLOCATED	-
1	1x	11000	FMINNMP (vector)	FEAT_AdvSIMD
1	1x	11010	FABD	FEAT_AdvSIMD
1	1x	11100	FCMGT (register)	FEAT_AdvSIMD
1	1x	11101	FACGT	FEAT_AdvSIMD
1	1x	11110	FMINP (vector)	FEAT_AdvSIMD
1	10	00011	BIT	FEAT_AdvSIMD
1	10	11001	FMLSL, FMLSL2 (vector) — FMLSL2	FEAT_FHM
1	11	00011	BIF	FEAT_AdvSIMD

C4.1.5.25 Advanced SIMD modified immediate

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

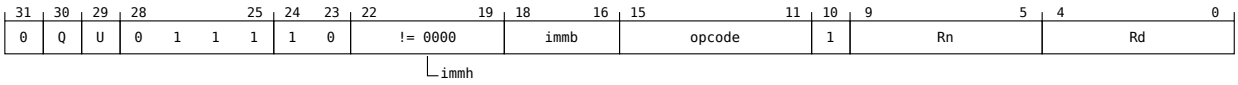
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential



Q	op	cmode	o2	Instruction Details	Feature
	0	0xx0	0	MOVI — 32-bit shifted immediate	FEAT_AdvSIMD
	0	0xx1	0	ORR (vector, immediate) — 32-bit	FEAT_AdvSIMD
	0	10x0	0	MOVI — 16-bit shifted immediate	FEAT_AdvSIMD
	0	10x1	0	ORR (vector, immediate) — 16-bit	FEAT_AdvSIMD
	0	110x	0	MOVI — 32-bit shifting ones	FEAT_AdvSIMD
	0	1110	0	MOVI — 8-bit	FEAT_AdvSIMD
	0	1111	0	FMOV (vector, immediate) — Single-precision	FEAT_AdvSIMD
	0	1111	1	FMOV (vector, immediate) — Half-precision	FEAT_AdvSIMD && FEAT_FP16
	0	!= 1111	1	UNALLOCATED	-
	1		1	UNALLOCATED	-
	1	0xx0	0	MVNI — 32-bit shifted immediate	FEAT_AdvSIMD
	1	0xx1	0	BIC (vector, immediate) — 32-bit	FEAT_AdvSIMD
	1	10x0	0	MVNI — 16-bit shifted immediate	FEAT_AdvSIMD
	1	10x1	0	BIC (vector, immediate) — 16-bit	FEAT_AdvSIMD
	1	110x	0	MVNI — 32-bit shifting ones	FEAT_AdvSIMD
0	1	1110	0	MOVI — 64-bit scalar	FEAT_AdvSIMD
0	1	1111	0	UNALLOCATED	-
1	1	1110	0	MOVI — 64-bit vector	FEAT_AdvSIMD
1	1	1111	0	FMOV (vector, immediate) — Double-precision	FEAT_AdvSIMD

C4.1.5.26 Advanced SIMD shift by immediate

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.



The following constraints also apply to this encoding: immh != ’0000’

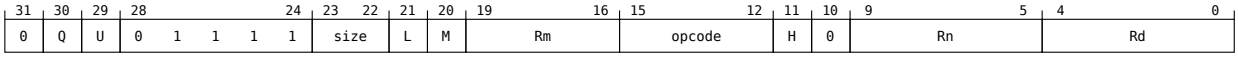
U	immh	opcode	Instruction Details	Feature
	!= 0000	0xxx1	UNALLOCATED	-
	!= 0000	1x110	UNALLOCATED	-
	!= 0000	101x1	UNALLOCATED	-
	!= 0000	110xx	UNALLOCATED	-

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	immh	opcode	Instruction Details	Feature
	!= 0000	11101	UNALLOCATED	-
0	!= 0000	00000	SSHR	FEAT_AdvSIMD
0	!= 0000	00010	SSRA	FEAT_AdvSIMD
0	!= 0000	00100	SRSHR	FEAT_AdvSIMD
0	!= 0000	00110	SRSRA	FEAT_AdvSIMD
0	!= 0000	01x00	UNALLOCATED	-
0	!= 0000	01010	SHL	FEAT_AdvSIMD
0	!= 0000	01110	SQSHL (immediate)	FEAT_AdvSIMD
0	!= 0000	10000	SHRN, SHRN2	FEAT_AdvSIMD
0	!= 0000	10001	RSHRN, RSHRN2	FEAT_AdvSIMD
0	!= 0000	10010	SQSHRN, SQSHRN2	FEAT_AdvSIMD
0	!= 0000	10011	SQRSHRN, SQRSHRN2	FEAT_AdvSIMD
0	!= 0000	10100	SSHLL, SSHLL2	FEAT_AdvSIMD
0	!= 0000	11100	SCVTF (vector, fixed-point)	FEAT_AdvSIMD
0	!= 0000	11111	FCVTZS (vector, fixed-point)	FEAT_AdvSIMD
1	!= 0000	00000	USHR	FEAT_AdvSIMD
1	!= 0000	00010	USRA	FEAT_AdvSIMD
1	!= 0000	00100	URSHR	FEAT_AdvSIMD
1	!= 0000	00110	URSRA	FEAT_AdvSIMD
1	!= 0000	01000	SRI	FEAT_AdvSIMD
1	!= 0000	01010	SLI	FEAT_AdvSIMD
1	!= 0000	01100	SQSHLU	FEAT_AdvSIMD
1	!= 0000	01110	UQSHL (immediate)	FEAT_AdvSIMD
1	!= 0000	10000	SQSHRUN, SQSHRUN2	FEAT_AdvSIMD
1	!= 0000	10001	SQRSHRUN, SQRSHRUN2	FEAT_AdvSIMD
1	!= 0000	10010	UQSHRN, UQSHRN2	FEAT_AdvSIMD
1	!= 0000	10011	UQRSHRN, UQRSHRN2	FEAT_AdvSIMD
1	!= 0000	10100	USHLL, USHLL2	FEAT_AdvSIMD
1	!= 0000	11100	UCVTF (vector, fixed-point)	FEAT_AdvSIMD
1	!= 0000	11111	FCVTZU (vector, fixed-point)	FEAT_AdvSIMD

C4.1.5.27 Advanced SIMD vector x indexed element

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.



Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

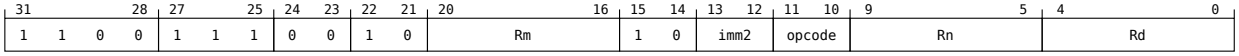
U	size	opcode	Instruction Details	Feature
	01	1001	UNALLOCATED	-
0		0010	SMLAL, SMLAL2 (by element)	FEAT_AdvSIMD
0		0011	SQDMLAL, SQDMLAL2 (by element)	FEAT_AdvSIMD
0		0110	SMLSL, SMLSL2 (by element)	FEAT_AdvSIMD
0		0111	SQDMLSL, SQDMLSL2 (by element)	FEAT_AdvSIMD
0		1000	MUL (by element)	FEAT_AdvSIMD
0		1010	SMULL, SMULL2 (by element)	FEAT_AdvSIMD
0		1011	SQDMULL, SQDMULL2 (by element)	FEAT_AdvSIMD
0		1100	SQDMULH (by element)	FEAT_AdvSIMD
0		1101	SQRDMULH (by element)	FEAT_AdvSIMD
0		1110	SDOT (by element)	FEAT_DotProd
0		1111	UNALLOCATED	-
0	00	0x00	UNALLOCATED	-
0	00	0001	FMLA (by element) — Vector, half-precision	FEAT_AdvSIMD && FEAT_FP16
0	00	0101	FMLS (by element) — Vector, half-precision	FEAT_AdvSIMD && FEAT_FP16
0	00	1001	FMUL (by element) — Vector, half-precision	FEAT_AdvSIMD && FEAT_FP16
0	01	0x0x	UNALLOCATED	-
0	1x	0001	FMLA (by element) — Vector, single-precision and double-precision	FEAT_AdvSIMD
0	1x	0101	FMLS (by element) — Vector, single-precision and double-precision	FEAT_AdvSIMD
0	1x	1001	FMUL (by element) — Vector, single-precision and double-precision	FEAT_AdvSIMD
0	10	0000	FMLAL, FMLAL2 (by element) — FMLAL	FEAT_FHM
0	10	0100	FMLSL, FMLSL2 (by element) — FMLSL	FEAT_FHM
0	11	0x00	UNALLOCATED	-
1		0xx1	FCMLA (by element)	FEAT_FCMA
1		0000	MLA (by element)	FEAT_AdvSIMD
1		0010	UMLAL, UMLAL2 (by element)	FEAT_AdvSIMD
1		0100	MLS (by element)	FEAT_AdvSIMD
1		0110	UMLSL, UMLSL2 (by element)	FEAT_AdvSIMD
1		1010	UMULL, UMULL2 (by element)	FEAT_AdvSIMD
1		1011	UNALLOCATED	-
1		1101	SQRDMLAH (by element)	FEAT_RDM
1		1110	UDOT (by element)	FEAT_DotProd
1		1111	SQRDMLSH (by element)	FEAT_RDM
1	00	1001	FMULX (by element) — Vector, half-precision	FEAT_AdvSIMD && FEAT_FP16

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	size	opcode	Instruction Details	Feature
1	1x	1001	FMULX (by element) — Vector, single-precision and double-precision	FEAT_AdvSIMD
1	10	1000	FMLAL, FMLAL2 (by element) — FMLAL2	FEAT_FHM
1	10	1100	FMLSL, FMLSL2 (by element) — FMLSL2	FEAT_FHM
1	!= 10	1x00	UNALLOCATED	-

C4.1.5.28 Cryptographic three-register, imm2

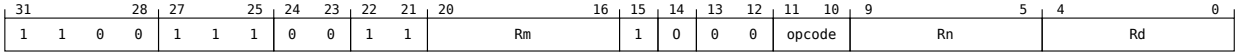
The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.



opcode	Instruction Details	Feature
00	SM3TT1A	FEAT_SM3
01	SM3TT1B	FEAT_SM3
10	SM3TT2A	FEAT_SM3
11	SM3TT2B	FEAT_SM3

C4.1.5.29 Cryptographic three-register SHA 512

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

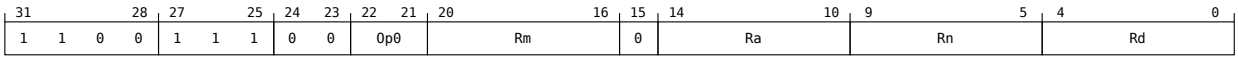


O	opcode	Instruction Details	Feature
0	00	SHA512H	FEAT_SHA512
0	01	SHA512H2	FEAT_SHA512
0	10	SHA512SU1	FEAT_SHA512
0	11	RAX1	FEAT_SHA3
1	00	SM3PARTW1	FEAT_SM3
1	01	SM3PARTW2	FEAT_SM3
1	10	SM4EKEY	FEAT_SM4
1	11	UNALLOCATED	-

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C4.1.5.30 Cryptographic four-register

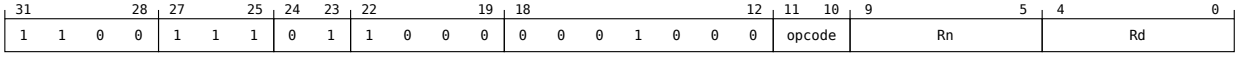
The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.



Op0	Instruction Details	Feature
00	EOR3	FEAT_SHA3
01	BCAX	FEAT_SHA3
10	SM3SS1	FEAT_SM3
11	UNALLOCATED	-

C4.1.5.31 Cryptographic two-register SHA 512

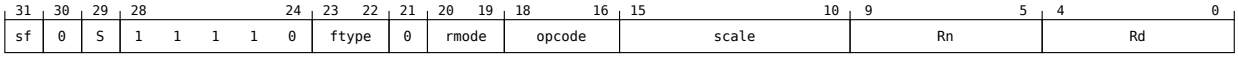
The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.



opcode	Instruction Details	Feature
00	SHA512SU0	FEAT_SHA512
01	SM4E	FEAT_SM4
1x	UNALLOCATED	-

C4.1.5.32 Conversion between floating-point and fixed-point

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.



sf	S	ftype	rmode	opcode	Instruction Details	Feature
	0			1xx	UNALLOCATED	-
	0	10		0xx	UNALLOCATED	-
	0	!= 10	x1	01x	UNALLOCATED	-
	0	!= 10	0x	00x	UNALLOCATED	-
	0	!= 10	10	0xx	UNALLOCATED	-
	1				UNALLOCATED	-
0	0	00	00	010	SCVTF (scalar, fixed-point) — 32-bit to single-precision	FEAT_FP

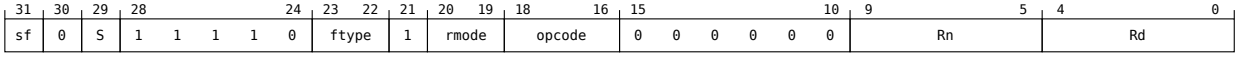
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sf	S	ftype	rmode	opcode	Instruction Details	Feature
0	0	00	00	011	UCVTF (scalar, fixed-point) — 32-bit to single-precision	FEAT_FP
0	0	00	11	000	FCVTZS (scalar, fixed-point) — Single-precision to 32-bit	FEAT_FP
0	0	00	11	001	FCVTZU (scalar, fixed-point) — Single-precision to 32-bit	FEAT_FP
0	0	01	00	010	SCVTF (scalar, fixed-point) — 32-bit to double-precision	FEAT_FP
0	0	01	00	011	UCVTF (scalar, fixed-point) — 32-bit to double-precision	FEAT_FP
0	0	01	11	000	FCVTZS (scalar, fixed-point) — Double-precision to 32-bit	FEAT_FP
0	0	01	11	001	FCVTZU (scalar, fixed-point) — Double-precision to 32-bit	FEAT_FP
0	0	11	00	010	SCVTF (scalar, fixed-point) — 32-bit to half-precision	FEAT_FP16
0	0	11	00	011	UCVTF (scalar, fixed-point) — 32-bit to half-precision	FEAT_FP16
0	0	11	11	000	FCVTZS (scalar, fixed-point) — Half-precision to 32-bit	FEAT_FP16
0	0	11	11	001	FCVTZU (scalar, fixed-point) — Half-precision to 32-bit	FEAT_FP16
1	0	00	00	010	SCVTF (scalar, fixed-point) — 64-bit to single-precision	FEAT_FP
1	0	00	00	011	UCVTF (scalar, fixed-point) — 64-bit to single-precision	FEAT_FP
1	0	00	11	000	FCVTZS (scalar, fixed-point) — Single-precision to 64-bit	FEAT_FP
1	0	00	11	001	FCVTZU (scalar, fixed-point) — Single-precision to 64-bit	FEAT_FP
1	0	01	00	010	SCVTF (scalar, fixed-point) — 64-bit to double-precision	FEAT_FP
1	0	01	00	011	UCVTF (scalar, fixed-point) — 64-bit to double-precision	FEAT_FP
1	0	01	11	000	FCVTZS (scalar, fixed-point) — Double-precision to 64-bit	FEAT_FP
1	0	01	11	001	FCVTZU (scalar, fixed-point) — Double-precision to 64-bit	FEAT_FP
1	0	11	00	010	SCVTF (scalar, fixed-point) — 64-bit to half-precision	FEAT_FP16
1	0	11	00	011	UCVTF (scalar, fixed-point) — 64-bit to half-precision	FEAT_FP16
1	0	11	11	000	FCVTZS (scalar, fixed-point) — Half-precision to 64-bit	FEAT_FP16
1	0	11	11	001	FCVTZU (scalar, fixed-point) — Half-precision to 64-bit	FEAT_FP16

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C4.1.5.33 Conversion between floating-point and integer

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.



sf	S	ftype	rmode	opcode	Instruction Details	Feature
	0		01	10x	UNALLOCATED	-
	0		11	10x	UNALLOCATED	-
	0	0x	x1	01x	UNALLOCATED	-
	0	0x	10	x1x	UNALLOCATED	-
	0	0x	11	111	UNALLOCATED	-
	0	00	11	110	UNALLOCATED	-
	0	10	x0		UNALLOCATED	-
	0	10	x1	0xx	UNALLOCATED	-
	0	10	11	11x	UNALLOCATED	-
	0	11	01	01x	UNALLOCATED	-
	0	11	1x	x1x	UNALLOCATED	-
	0	!= 10	10	10x	UNALLOCATED	-
	1				UNALLOCATED	-
0	0		01	11x	UNALLOCATED	-
0	0	00	00	000	FCVTNS (scalar) — Single-precision to 32-bit	FEAT_FP
0	0	00	00	001	FCVTNU (scalar) — Single-precision to 32-bit	FEAT_FP
0	0	00	00	010	SCVTF (scalar, integer) — 32-bit to single-precision	FEAT_FP
0	0	00	00	011	UCVTF (scalar, integer) — 32-bit to single-precision	FEAT_FP
0	0	00	00	100	FCVTAS (scalar) — Single-precision to 32-bit	FEAT_FP
0	0	00	00	101	FCVTAU (scalar) — Single-precision to 32-bit	FEAT_FP
0	0	00	00	110	FMOV (general) — Single-precision to 32-bit	FEAT_FP
0	0	00	00	111	FMOV (general) — 32-bit to single-precision	FEAT_FP
0	0	00	01	000	FCVTPS (scalar) — Single-precision to 32-bit	FEAT_FP
0	0	00	01	001	FCVTPU (scalar) — Single-precision to 32-bit	FEAT_FP
0	0	00	10	000	FCVTMS (scalar) — Single-precision to 32-bit	FEAT_FP
0	0	00	10	001	FCVTMU (scalar) — Single-precision to 32-bit	FEAT_FP

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sf	S	ftype	rmode	opcode	Instruction Details	Feature
0	0	00	11	000	FCVTZS (scalar, integer) — Single-precision to 32-bit	FEAT_FP
0	0	00	11	001	FCVTZU (scalar, integer) — Single-precision to 32-bit	FEAT_FP
0	0	01	00	000	FCVTNS (scalar) — Double-precision to 32-bit	FEAT_FP
0	0	01	00	001	FCVTNU (scalar) — Double-precision to 32-bit	FEAT_FP
0	0	01	00	010	SCVTF (scalar, integer) — 32-bit to double-precision	FEAT_FP
0	0	01	00	011	UCVTF (scalar, integer) — 32-bit to double-precision	FEAT_FP
0	0	01	00	100	FCVTAS (scalar) — Double-precision to 32-bit	FEAT_FP
0	0	01	00	101	FCVTAU (scalar) — Double-precision to 32-bit	FEAT_FP
0	0	01	00	11x	UNALLOCATED	-
0	0	01	01	000	FCVTPS (scalar) — Double-precision to 32-bit	FEAT_FP
0	0	01	01	001	FCVTPU (scalar) — Double-precision to 32-bit	FEAT_FP
0	0	01	10	000	FCVTMS (scalar) — Double-precision to 32-bit	FEAT_FP
0	0	01	10	001	FCVTMU (scalar) — Double-precision to 32-bit	FEAT_FP
0	0	01	11	000	FCVTZS (scalar, integer) — Double-precision to 32-bit	FEAT_FP
0	0	01	11	001	FCVTZU (scalar, integer) — Double-precision to 32-bit	FEAT_FP
0	0	01	11	110	FJCVTZS	FEAT_JSCVT
0	0	11	00	000	FCVTNS (scalar) — Half-precision to 32-bit	FEAT_FP16
0	0	11	00	001	FCVTNU (scalar) — Half-precision to 32-bit	FEAT_FP16
0	0	11	00	010	SCVTF (scalar, integer) — 32-bit to half-precision	FEAT_FP16
0	0	11	00	011	UCVTF (scalar, integer) — 32-bit to half-precision	FEAT_FP16
0	0	11	00	100	FCVTAS (scalar) — Half-precision to 32-bit	FEAT_FP16
0	0	11	00	101	FCVTAU (scalar) — Half-precision to 32-bit	FEAT_FP16
0	0	11	00	110	FMOV (general) — Half-precision to 32-bit	FEAT_FP16
0	0	11	00	111	FMOV (general) — 32-bit to half-precision	FEAT_FP16
0	0	11	01	000	FCVTPS (scalar) — Half-precision to 32-bit	FEAT_FP16
0	0	11	01	001	FCVTPU (scalar) — Half-precision to 32-bit	FEAT_FP16

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sf	S	ftype	rmode	opcode	Instruction Details	Feature
0	0	11	10	000	FCVTMS (scalar) — Half-precision to 32-bit	FEAT_FP16
0	0	11	10	001	FCVTMU (scalar) — Half-precision to 32-bit	FEAT_FP16
0	0	11	11	000	FCVTZS (scalar, integer) — Half-precision to 32-bit	FEAT_FP16
0	0	11	11	001	FCVTZU (scalar, integer) — Half-precision to 32-bit	FEAT_FP16
1	0	00	00	000	FCVTNS (scalar) — Single-precision to 64-bit	FEAT_FP
1	0	00	00	001	FCVTNU (scalar) — Single-precision to 64-bit	FEAT_FP
1	0	00	00	010	SCVTF (scalar, integer) — 64-bit to single-precision	FEAT_FP
1	0	00	00	011	UCVTF (scalar, integer) — 64-bit to single-precision	FEAT_FP
1	0	00	00	100	FCVTAS (scalar) — Single-precision to 64-bit	FEAT_FP
1	0	00	00	101	FCVTAU (scalar) — Single-precision to 64-bit	FEAT_FP
1	0	00	00	11x	UNALLOCATED	-
1	0	00	01	000	FCVTPS (scalar) — Single-precision to 64-bit	FEAT_FP
1	0	00	01	001	FCVTPU (scalar) — Single-precision to 64-bit	FEAT_FP
1	0	00	10	000	FCVTMS (scalar) — Single-precision to 64-bit	FEAT_FP
1	0	00	10	001	FCVTMU (scalar) — Single-precision to 64-bit	FEAT_FP
1	0	00	11	000	FCVTZS (scalar, integer) — Single-precision to 64-bit	FEAT_FP
1	0	00	11	001	FCVTZU (scalar, integer) — Single-precision to 64-bit	FEAT_FP
1	0	01	00	000	FCVTNS (scalar) — Double-precision to 64-bit	FEAT_FP
1	0	01	00	001	FCVTNU (scalar) — Double-precision to 64-bit	FEAT_FP
1	0	01	00	010	SCVTF (scalar, integer) — 64-bit to double-precision	FEAT_FP
1	0	01	00	011	UCVTF (scalar, integer) — 64-bit to double-precision	FEAT_FP
1	0	01	00	100	FCVTAS (scalar) — Double-precision to 64-bit	FEAT_FP
1	0	01	00	101	FCVTAU (scalar) — Double-precision to 64-bit	FEAT_FP
1	0	01	00	110	FMOV (general) — Double-precision to 64-bit	FEAT_FP
1	0	01	00	111	FMOV (general) — 64-bit to double-precision	FEAT_FP
1	0	01	01	000	FCVTPS (scalar) — Double-precision to 64-bit	FEAT_FP

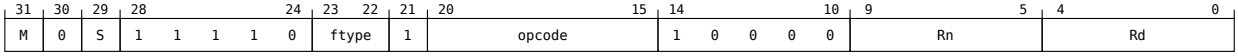
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sf	S	ftype	rmode	opcode	Instruction Details	Feature
1	0	01	01	001	FCVTPU (scalar) — Double-precision to 64-bit	FEAT_FP
1	0	01	10	000	FCVTMS (scalar) — Double-precision to 64-bit	FEAT_FP
1	0	01	10	001	FCVTMU (scalar) — Double-precision to 64-bit	FEAT_FP
1	0	01	11	000	FCVTZS (scalar, integer) — Double-precision to 64-bit	FEAT_FP
1	0	01	11	001	FCVTZU (scalar, integer) — Double-precision to 64-bit	FEAT_FP
1	0	01	11	110	UNALLOCATED	-
1	0	10	01	110	FMOV (general) — Top half of 128-bit to 64-bit	FEAT_FP
1	0	10	01	111	FMOV (general) — 64-bit to top half of 128-bit	FEAT_FP
1	0	11	00	000	FCVTNS (scalar) — Half-precision to 64-bit	FEAT_FP16
1	0	11	00	001	FCVTNU (scalar) — Half-precision to 64-bit	FEAT_FP16
1	0	11	00	010	SCVTF (scalar, integer) — 64-bit to half-precision	FEAT_FP16
1	0	11	00	011	UCVTF (scalar, integer) — 64-bit to half-precision	FEAT_FP16
1	0	11	00	100	FCVTAS (scalar) — Half-precision to 64-bit	FEAT_FP16
1	0	11	00	101	FCVTAU (scalar) — Half-precision to 64-bit	FEAT_FP16
1	0	11	00	110	FMOV (general) — Half-precision to 64-bit	FEAT_FP16
1	0	11	00	111	FMOV (general) — 64-bit to half-precision	FEAT_FP16
1	0	11	01	000	FCVTPS (scalar) — Half-precision to 64-bit	FEAT_FP16
1	0	11	01	001	FCVTPU (scalar) — Half-precision to 64-bit	FEAT_FP16
1	0	11	10	000	FCVTMS (scalar) — Half-precision to 64-bit	FEAT_FP16
1	0	11	10	001	FCVTMU (scalar) — Half-precision to 64-bit	FEAT_FP16
1	0	11	11	000	FCVTZS (scalar, integer) — Half-precision to 64-bit	FEAT_FP16
1	0	11	11	001	FCVTZU (scalar, integer) — Half-precision to 64-bit	FEAT_FP16
1	0	!= 10	01	11x	UNALLOCATED	-

C4.1.5.34 Floating-point data-processing (1 source)

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential



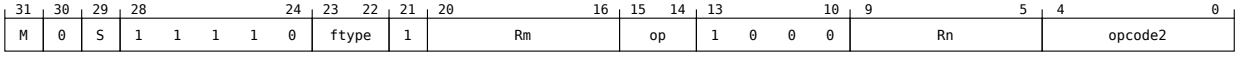
M	S	ftype	opcode	Instruction Details	Feature
0	0		01xxxx	UNALLOCATED	-
0	0		1xxxxx	UNALLOCATED	-
0	0	00	000000	FMOV (register) — Single-precision	FEAT_FP
0	0	00	000001	FABS (scalar) — Single-precision	FEAT_FP
0	0	00	000010	FNEG (scalar) — Single-precision	FEAT_FP
0	0	00	000011	FSQRT (scalar) — Single-precision	FEAT_FP
0	0	00	0001x0	UNALLOCATED	-
0	0	00	000101	FCVT — Single-precision to double-precision	FEAT_FP
0	0	00	000111	FCVT — Single-precision to half-precision	FEAT_FP
0	0	00	001000	FRINTN (scalar) — Single-precision	FEAT_FP
0	0	00	001001	FRINTP (scalar) — Single-precision	FEAT_FP
0	0	00	001010	FRINTM (scalar) — Single-precision	FEAT_FP
0	0	00	001011	FRINTZ (scalar) — Single-precision	FEAT_FP
0	0	00	001100	FRINTA (scalar) — Single-precision	FEAT_FP
0	0	00	001110	FRINTX (scalar) — Single-precision	FEAT_FP
0	0	00	001111	FRINTI (scalar) — Single-precision	FEAT_FP
0	0	01	000000	FMOV (register) — Double-precision	FEAT_FP
0	0	01	000001	FABS (scalar) — Double-precision	FEAT_FP
0	0	01	000010	FNEG (scalar) — Double-precision	FEAT_FP
0	0	01	000011	FSQRT (scalar) — Double-precision	FEAT_FP
0	0	01	000100	FCVT — Double-precision to single-precision	FEAT_FP
0	0	01	000101	UNALLOCATED	-
0	0	01	000110	UNALLOCATED	-
0	0	01	000111	FCVT — Double-precision to half-precision	FEAT_FP
0	0	01	001000	FRINTN (scalar) — Double-precision	FEAT_FP
0	0	01	001001	FRINTP (scalar) — Double-precision	FEAT_FP
0	0	01	001010	FRINTM (scalar) — Double-precision	FEAT_FP
0	0	01	001011	FRINTZ (scalar) — Double-precision	FEAT_FP
0	0	01	001100	FRINTA (scalar) — Double-precision	FEAT_FP
0	0	01	001110	FRINTX (scalar) — Double-precision	FEAT_FP
0	0	01	001111	FRINTI (scalar) — Double-precision	FEAT_FP
0	0	10	00xxxx	UNALLOCATED	-
0	0	11	000000	FMOV (register) — Half-precision	FEAT_FP16
0	0	11	000001	FABS (scalar) — Half-precision	FEAT_FP16

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

M	S	ftype	opcode	Instruction Details	Feature
0	0	11	000010	FNEG (scalar) — Half-precision	FEAT_FP16
0	0	11	000011	FSQRT (scalar) — Half-precision	FEAT_FP16
0	0	11	000100	FCVT — Half-precision to single-precision	FEAT_FP
0	0	11	000101	FCVT — Half-precision to double-precision	FEAT_FP
0	0	11	00011x	UNALLOCATED	-
0	0	11	001000	FRINTN (scalar) — Half-precision	FEAT_FP16
0	0	11	001001	FRINTP (scalar) — Half-precision	FEAT_FP16
0	0	11	001010	FRINTM (scalar) — Half-precision	FEAT_FP16
0	0	11	001011	FRINTZ (scalar) — Half-precision	FEAT_FP16
0	0	11	001100	FRINTA (scalar) — Half-precision	FEAT_FP16
0	0	11	001110	FRINTX (scalar) — Half-precision	FEAT_FP16
0	0	11	001111	FRINTI (scalar) — Half-precision	FEAT_FP16
0	0	!= 10	001101	UNALLOCATED	-
0	1			UNALLOCATED	-
1				UNALLOCATED	-

C4.1.5.35 Floating-point compare

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.



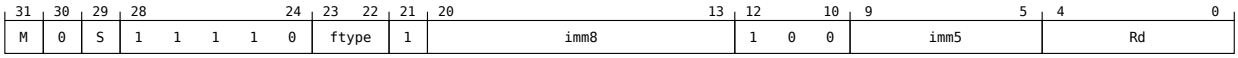
M	S	ftype	op	opcode2	Instruction Details	Feature
0	0		00	xx001	UNALLOCATED	-
0	0		00	xx01x	UNALLOCATED	-
0	0		00	xx1xx	UNALLOCATED	-
0	0		!= 00		UNALLOCATED	-
0	0	00	00	00000	FCMP — Single-precision	FEAT_FP
0	0	00	00	01000	FCMP — Single-precision, zero	FEAT_FP
0	0	00	00	10000	FCMPE — Single-precision	FEAT_FP
0	0	00	00	11000	FCMPE — Single-precision, zero	FEAT_FP
0	0	01	00	00000	FCMP — Double-precision	FEAT_FP
0	0	01	00	01000	FCMP — Double-precision, zero	FEAT_FP
0	0	01	00	10000	FCMPE — Double-precision	FEAT_FP
0	0	01	00	11000	FCMPE — Double-precision, zero	FEAT_FP
0	0	10	00	xx000	UNALLOCATED	-
0	0	11	00	00000	FCMP — Half-precision	FEAT_FP16

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

M	S	ftype	op	opcode2	Instruction Details	Feature
0	0	11	00	01000	FCMP — Half-precision, zero	FEAT_FP16
0	0	11	00	10000	FCMPE — Half-precision	FEAT_FP16
0	0	11	00	11000	FCMPE — Half-precision, zero	FEAT_FP16
0	1				UNALLOCATED	-
1					UNALLOCATED	-

C4.1.5.36 Floating-point immediate

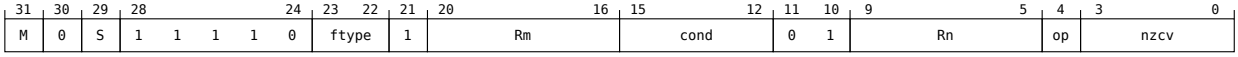
The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.



M	S	ftype	imm5	Instruction Details	Feature
0	0		!= 00000	UNALLOCATED	-
0	0	00	00000	FMOV (scalar, immediate) — Single-precision	FEAT_FP
0	0	01	00000	FMOV (scalar, immediate) — Double-precision	FEAT_FP
0	0	10	00000	UNALLOCATED	-
0	0	11	00000	FMOV (scalar, immediate) — Half-precision	FEAT_FP16
0	1			UNALLOCATED	-
1				UNALLOCATED	-

C4.1.5.37 Floating-point conditional compare

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.



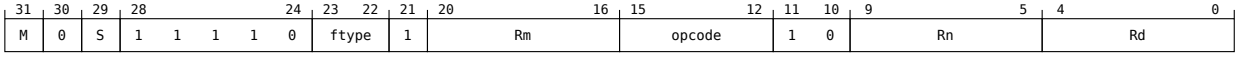
M	S	ftype	op	Instruction Details	Feature
0	0	00	0	FCCMP — Single-precision	FEAT_FP
0	0	00	1	FCCMPE — Single-precision	FEAT_FP
0	0	01	0	FCCMP — Double-precision	FEAT_FP
0	0	01	1	FCCMPE — Double-precision	FEAT_FP
0	0	10		UNALLOCATED	-
0	0	11	0	FCCMP — Half-precision	FEAT_FP16
0	0	11	1	FCCMPE — Half-precision	FEAT_FP16

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

M	S	ftype	op	Instruction Details	Feature
0	1			UNALLOCATED	-
1				UNALLOCATED	-

C4.1.5.38 Floating-point data-processing (2 source)

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.



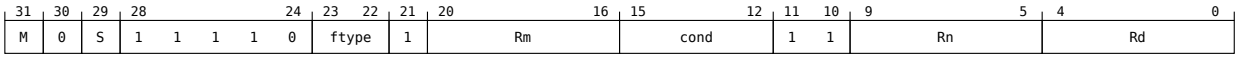
M	S	ftype	opcode	Instruction Details	Feature
0	0	00	0000	FMUL (scalar) — Single-precision	FEAT_FP
0	0	00	0001	FDIV (scalar) — Single-precision	FEAT_FP
0	0	00	0010	FADD (scalar) — Single-precision	FEAT_FP
0	0	00	0011	FSUB (scalar) — Single-precision	FEAT_FP
0	0	00	0100	FMAX (scalar) — Single-precision	FEAT_FP
0	0	00	0101	FMIN (scalar) — Single-precision	FEAT_FP
0	0	00	0110	FMAXNM (scalar) — Single-precision	FEAT_FP
0	0	00	0111	FMINNM (scalar) — Single-precision	FEAT_FP
0	0	00	1000	FNMUL (scalar) — Single-precision	FEAT_FP
0	0	01	0000	FMUL (scalar) — Double-precision	FEAT_FP
0	0	01	0001	FDIV (scalar) — Double-precision	FEAT_FP
0	0	01	0010	FADD (scalar) — Double-precision	FEAT_FP
0	0	01	0011	FSUB (scalar) — Double-precision	FEAT_FP
0	0	01	0100	FMAX (scalar) — Double-precision	FEAT_FP
0	0	01	0101	FMIN (scalar) — Double-precision	FEAT_FP
0	0	01	0110	FMAXNM (scalar) — Double-precision	FEAT_FP
0	0	01	0111	FMINNM (scalar) — Double-precision	FEAT_FP
0	0	01	1000	FNMUL (scalar) — Double-precision	FEAT_FP
0	0	10		UNALLOCATED	-
0	0	11	0000	FMUL (scalar) — Half-precision	FEAT_FP16
0	0	11	0001	FDIV (scalar) — Half-precision	FEAT_FP16
0	0	11	0010	FADD (scalar) — Half-precision	FEAT_FP16
0	0	11	0011	FSUB (scalar) — Half-precision	FEAT_FP16
0	0	11	0100	FMAX (scalar) — Half-precision	FEAT_FP16
0	0	11	0101	FMIN (scalar) — Half-precision	FEAT_FP16
0	0	11	0110	FMAXNM (scalar) — Half-precision	FEAT_FP16
0	0	11	0111	FMINNM (scalar) — Half-precision	FEAT_FP16
0	0	11	1000	FNMUL (scalar) — Half-precision	FEAT_FP16

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

M	S	ftype	opcode	Instruction Details	Feature
0	0	!= 10	1001	UNALLOCATED	-
0	0	!= 10	101x	UNALLOCATED	-
0	0	!= 10	11xx	UNALLOCATED	-
0	1			UNALLOCATED	-
1				UNALLOCATED	-

C4.1.5.39 Floating-point conditional select

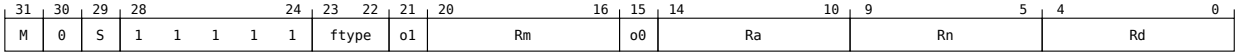
The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.



M	S	ftype	Instruction Details	Feature
0	0	00	FCSEL — Single-precision	FEAT_FP
0	0	01	FCSEL — Double-precision	FEAT_FP
0	0	10	UNALLOCATED	-
0	0	11	FCSEL — Half-precision	FEAT_FP16
0	1		UNALLOCATED	-
1			UNALLOCATED	-

C4.1.5.40 Floating-point data-processing (3 source)

The encodings in this section are decoded from Data Processing – Scalar Floating-Point and Advanced SIMD.



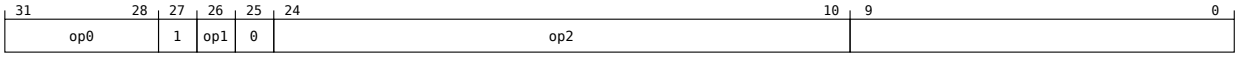
M	S	ftype	o1	o0	Instruction Details	Feature
0	0	00	0	0	FMADD — Single-precision	FEAT_FP
0	0	00	0	1	FMSUB — Single-precision	FEAT_FP
0	0	00	1	0	FNMADD — Single-precision	FEAT_FP
0	0	00	1	1	FNMSUB — Single-precision	FEAT_FP
0	0	01	0	0	FMADD — Double-precision	FEAT_FP
0	0	01	0	1	FMSUB — Double-precision	FEAT_FP
0	0	01	1	0	FNMADD — Double-precision	FEAT_FP
0	0	01	1	1	FNMSUB — Double-precision	FEAT_FP
0	0	10			UNALLOCATED	-
0	0	11	0	0	FMADD — Half-precision	FEAT_FP16

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

M	S	ftype	o1	o0	Instruction Details	Feature
0	0	11	0	1	FMSUB — Half-precision	FEAT_FP16
0	0	11	1	0	FNMADD — Half-precision	FEAT_FP16
0	0	11	1	1	FNMSUB — Half-precision	FEAT_FP16
0	1				UNALLOCATED	-
1					UNALLOCATED	-

C4.1.6 Loads and Stores

The encodings in this section are decoded from A64 instruction set encoding.



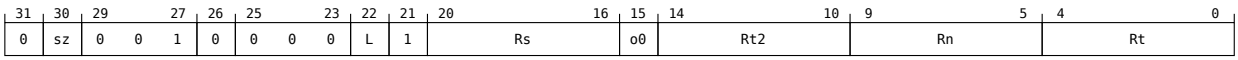
op0	op1	op2	Instruction details
0x00	0	00x1xxxxxxxxxx	Compare and swap pair
0x00	1	00x000000xxxxxx	Advanced SIMD load/store multiple structures
0x00	1	00x000001xxxxxx	UNALLOCATED
0x00	1	01x0xxxxxxxxxx	Advanced SIMD load/store multiple structures (post-indexed)
0x00	1	0xx1xxxxxxxxxx	UNALLOCATED
0x00	1	10x10001xxxxxxx	UNALLOCATED
0x00	1	10x1001xxxxxxxx	UNALLOCATED
0x00	1	10x101xxxxxxxxx	UNALLOCATED
0x00	1	10x11xxxxxxxxxx	UNALLOCATED
0x00	1	10xx0000xxxxxxx	Advanced SIMD load/store single structure
0x00	1	11xxxxxxxxxxxx	Advanced SIMD load/store single structure (post-indexed)
0x00	1	x0x00001xxxxxxx	UNALLOCATED
0x00	1	x0x0001xxxxxxxx	UNALLOCATED
0x00	1	x0x001xxxxxxxxx	UNALLOCATED
0x00	1	x0x01xxxxxxxxx	UNALLOCATED
1x00	0	00x1xxxxxxxxxx	Load/store exclusive pair
1x00	1		UNALLOCATED
xx00	0	00x0xxxxxxxxxx	Load/store exclusive register
xx00	0	01x0xxxxxxxxxx	Load/store ordered
xx00	0	01x1xxxxxxxxxx	Compare and swap
xx00	0	1xxxxxxxxxxxxx	UNALLOCATED
xx01	0	1xx0xxxxxxxx00	Load/store ordered (unscaled immediate)
xx01	0	1xx0xxxxxxxx01	UNALLOCATED

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	op2	Instruction details
xx01	0	1xx0xxxxxxxx1x	UNALLOCATED
xx01	0	1xx1xxxxxxxxxx	UNALLOCATED
xx01	1	1xxxxxxxxxxxxx	UNALLOCATED
xx01		0xxxxxxxxxxxxx	Load register (literal)
xx10		00xxxxxxxxxxxx	Load/store no-allocate pair (offset)
xx10		01xxxxxxxxxxxx	Load/store register pair (post-indexed)
xx10		10xxxxxxxxxxxx	Load/store register pair (offset)
xx10		11xxxxxxxxxxxx	Load/store register pair (pre-indexed)
xx11		0xx0xxxxxxxx00	Load/store register (unscaled immediate)
xx11		0xx0xxxxxxxx01	Load/store register (immediate post-indexed)
xx11		0xx0xxxxxxxx10	Load/store register (unprivileged)
xx11		0xx0xxxxxxxx11	Load/store register (immediate pre-indexed)
xx11		0xx1xxxxxxxx00	Atomic memory operations
xx11		0xx1xxxxxxxx10	Load/store register (register offset)
xx11		0xx1xxxxxxxxx1	Load/store register (pac)
xx11		1xxxxxxxxxxxxx	Load/store register (unsigned immediate)

C4.1.6.1 Compare and swap pair

The encodings in this section are decoded from Loads and Stores.

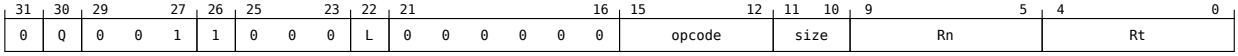


sz	L	o0	Rt2	Instruction Details	Feature
			!= 11111	UNALLOCATED	-
0	0	0	11111	CASP, CASPA, CASPAL, CASPL — 32-bit CASP	FEAT_LSE
0	0	1	11111	CASP, CASPA, CASPAL, CASPL — 32-bit CASPL	FEAT_LSE
0	1	0	11111	CASP, CASPA, CASPAL, CASPL — 32-bit CASPA	FEAT_LSE
0	1	1	11111	CASP, CASPA, CASPAL, CASPL — 32-bit CASPAL	FEAT_LSE
1	0	0	11111	CASP, CASPA, CASPAL, CASPL — 64-bit CASP	FEAT_LSE
1	0	1	11111	CASP, CASPA, CASPAL, CASPL — 64-bit CASPL	FEAT_LSE
1	1	0	11111	CASP, CASPA, CASPAL, CASPL — 64-bit CASPA	FEAT_LSE
1	1	1	11111	CASP, CASPA, CASPAL, CASPL — 64-bit CASPAL	FEAT_LSE

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C4.1.6.2 Advanced SIMD load/store multiple structures

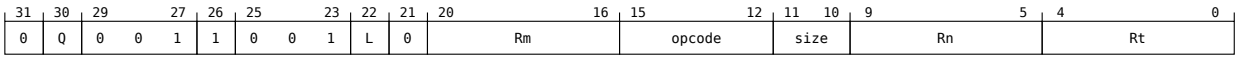
The encodings in this section are decoded from Loads and Stores.



L	opcode	Instruction Details	Feature
	x0x1	UNALLOCATED	-
	0101	UNALLOCATED	-
	11xx	UNALLOCATED	-
0	0000	ST4 (multiple structures)	FEAT_AdvSIMD
0	0010	ST1 (multiple structures) — Four registers	FEAT_AdvSIMD
0	0100	ST3 (multiple structures)	FEAT_AdvSIMD
0	0110	ST1 (multiple structures) — Three registers	FEAT_AdvSIMD
0	0111	ST1 (multiple structures) — One register	FEAT_AdvSIMD
0	1000	ST2 (multiple structures)	FEAT_AdvSIMD
0	1010	ST1 (multiple structures) — Two registers	FEAT_AdvSIMD
1	0000	LD4 (multiple structures)	FEAT_AdvSIMD
1	0010	LD1 (multiple structures) — Four registers	FEAT_AdvSIMD
1	0100	LD3 (multiple structures)	FEAT_AdvSIMD
1	0110	LD1 (multiple structures) — Three registers	FEAT_AdvSIMD
1	0111	LD1 (multiple structures) — One register	FEAT_AdvSIMD
1	1000	LD2 (multiple structures)	FEAT_AdvSIMD
1	1010	LD1 (multiple structures) — Two registers	FEAT_AdvSIMD

C4.1.6.3 Advanced SIMD load/store multiple structures (post-indexed)

The encodings in this section are decoded from Loads and Stores.



L	Rm	opcode	Instruction Details	Feature
		x0x1	UNALLOCATED	-
		0101	UNALLOCATED	-

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

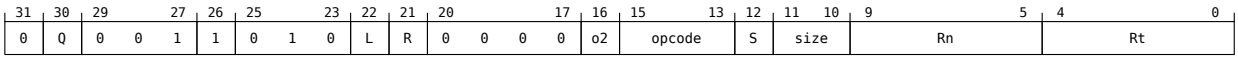
L	Rm	opcode	Instruction Details	Feature
		11xx	UNALLOCATED	-
0	11111	0000	ST4 (multiple structures) — Immediate offset	FEAT_AdvSIMD
0	11111	0010	ST1 (multiple structures) — Four registers, immediate offset	FEAT_AdvSIMD
0	11111	0100	ST3 (multiple structures) — Immediate offset	FEAT_AdvSIMD
0	11111	0110	ST1 (multiple structures) — Three registers, immediate offset	FEAT_AdvSIMD
0	11111	0111	ST1 (multiple structures) — One register, immediate offset	FEAT_AdvSIMD
0	11111	1000	ST2 (multiple structures) — Immediate offset	FEAT_AdvSIMD
0	11111	1010	ST1 (multiple structures) — Two registers, immediate offset	FEAT_AdvSIMD
0	!=11111	0000	ST4 (multiple structures) — Register offset	FEAT_AdvSIMD
0	!=11111	0010	ST1 (multiple structures) — Four registers, register offset	FEAT_AdvSIMD
0	!=11111	0100	ST3 (multiple structures) — Register offset	FEAT_AdvSIMD
0	!=11111	0110	ST1 (multiple structures) — Three registers, register offset	FEAT_AdvSIMD
0	!=11111	0111	ST1 (multiple structures) — One register, register offset	FEAT_AdvSIMD
0	!=11111	1000	ST2 (multiple structures) — Register offset	FEAT_AdvSIMD
0	!=11111	1010	ST1 (multiple structures) — Two registers, register offset	FEAT_AdvSIMD
1	11111	0000	LD4 (multiple structures) — Immediate offset	FEAT_AdvSIMD
1	11111	0010	LD1 (multiple structures) — Four registers, immediate offset	FEAT_AdvSIMD
1	11111	0100	LD3 (multiple structures) — Immediate offset	FEAT_AdvSIMD
1	11111	0110	LD1 (multiple structures) — Three registers, immediate offset	FEAT_AdvSIMD
1	11111	0111	LD1 (multiple structures) — One register, immediate offset	FEAT_AdvSIMD
1	11111	1000	LD2 (multiple structures) — Immediate offset	FEAT_AdvSIMD
1	11111	1010	LD1 (multiple structures) — Two registers, immediate offset	FEAT_AdvSIMD
1	!=11111	0000	LD4 (multiple structures) — Register offset	FEAT_AdvSIMD
1	!=11111	0010	LD1 (multiple structures) — Four registers, register offset	FEAT_AdvSIMD
1	!=11111	0100	LD3 (multiple structures) — Register offset	FEAT_AdvSIMD
1	!=11111	0110	LD1 (multiple structures) — Three registers, register offset	FEAT_AdvSIMD

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

L	Rm	opcode	Instruction Details	Feature
1	!= 11111	0111	LD1 (multiple structures) — One register, register offset	FEAT_AdvSIMD
1	!= 11111	1000	LD2 (multiple structures) — Register offset	FEAT_AdvSIMD
1	!= 11111	1010	LD1 (multiple structures) — Two registers, register offset	FEAT_AdvSIMD

C4.1.6.4 Advanced SIMD load/store single structure

The encodings in this section are decoded from Loads and Stores.



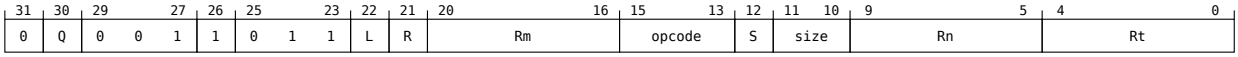
L	R	o2	opcode	S	size	Instruction Details	Feature
		0	01x		x1	UNALLOCATED	-
		0	10x		1x	UNALLOCATED	-
		0	10x	1	01	UNALLOCATED	-
		1				UNALLOCATED	-
0		0	11x			UNALLOCATED	-
0	0	0	000			ST1 (single structure) — 8-bit	FEAT_AdvSIMD
0	0	0	001			ST3 (single structure) — 8-bit	FEAT_AdvSIMD
0	0	0	010		x0	ST1 (single structure) — 16-bit	FEAT_AdvSIMD
0	0	0	011		x0	ST3 (single structure) — 16-bit	FEAT_AdvSIMD
0	0	0	100		00	ST1 (single structure) — 32-bit	FEAT_AdvSIMD
0	0	0	100	0	01	ST1 (single structure) — 64-bit	FEAT_AdvSIMD
0	0	0	101		00	ST3 (single structure) — 32-bit	FEAT_AdvSIMD
0	0	0	101	0	01	ST3 (single structure) — 64-bit	FEAT_AdvSIMD
0	1	0	000			ST2 (single structure) — 8-bit	FEAT_AdvSIMD
0	1	0	001			ST4 (single structure) — 8-bit	FEAT_AdvSIMD
0	1	0	010		x0	ST2 (single structure) — 16-bit	FEAT_AdvSIMD
0	1	0	011		x0	ST4 (single structure) — 16-bit	FEAT_AdvSIMD
0	1	0	100		00	ST2 (single structure) — 32-bit	FEAT_AdvSIMD
0	1	0	100	0	01	ST2 (single structure) — 64-bit	FEAT_AdvSIMD
0	1	0	101		00	ST4 (single structure) — 32-bit	FEAT_AdvSIMD
0	1	0	101	0	01	ST4 (single structure) — 64-bit	FEAT_AdvSIMD
1		0	11x	1		UNALLOCATED	-
1	0	0	000			LD1 (single structure) — 8-bit	FEAT_AdvSIMD
1	0	0	001			LD3 (single structure) — 8-bit	FEAT_AdvSIMD
1	0	0	010		x0	LD1 (single structure) — 16-bit	FEAT_AdvSIMD
1	0	0	011		x0	LD3 (single structure) — 16-bit	FEAT_AdvSIMD

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

L	R	o2	opcode	S	size	Instruction Details	Feature
1	0	0	100		00	LD1 (single structure) — 32-bit	FEAT_AdvSIMD
1	0	0	100	0	01	LD1 (single structure) — 64-bit	FEAT_AdvSIMD
1	0	0	101		00	LD3 (single structure) — 32-bit	FEAT_AdvSIMD
1	0	0	101	0	01	LD3 (single structure) — 64-bit	FEAT_AdvSIMD
1	0	0	110	0		LD1R	FEAT_AdvSIMD
1	0	0	111	0		LD3R	FEAT_AdvSIMD
1	1	0	000			LD2 (single structure) — 8-bit	FEAT_AdvSIMD
1	1	0	001			LD4 (single structure) — 8-bit	FEAT_AdvSIMD
1	1	0	010		x0	LD2 (single structure) — 16-bit	FEAT_AdvSIMD
1	1	0	011		x0	LD4 (single structure) — 16-bit	FEAT_AdvSIMD
1	1	0	100		00	LD2 (single structure) — 32-bit	FEAT_AdvSIMD
1	1	0	100	0	01	LD2 (single structure) — 64-bit	FEAT_AdvSIMD
1	1	0	101		00	LD4 (single structure) — 32-bit	FEAT_AdvSIMD
1	1	0	101	0	01	LD4 (single structure) — 64-bit	FEAT_AdvSIMD
1	1	0	110	0		LD2R	FEAT_AdvSIMD
1	1	0	111	0		LD4R	FEAT_AdvSIMD

C4.1.6.5 Advanced SIMD load/store single structure (post-indexed)

The encodings in this section are decoded from Loads and Stores.



L	R	Rm	opcode	S	size	Instruction Details	Feature
			01x		x1	UNALLOCATED	-
			10x		1x	UNALLOCATED	-
			10x	1	01	UNALLOCATED	-
0			11x			UNALLOCATED	-
0	0	11111	000			ST1 (single structure) — 8-bit, immediate offset	FEAT_AdvSIMD
0	0	11111	001			ST3 (single structure) — 8-bit, immediate offset	FEAT_AdvSIMD
0	0	11111	010		x0	ST1 (single structure) — 16-bit, immediate offset	FEAT_AdvSIMD
0	0	11111	011		x0	ST3 (single structure) — 16-bit, immediate offset	FEAT_AdvSIMD
0	0	11111	100		00	ST1 (single structure) — 32-bit, immediate offset	FEAT_AdvSIMD
0	0	11111	100	0	01	ST1 (single structure) — 64-bit, immediate offset	FEAT_AdvSIMD
0	0	11111	101		00	ST3 (single structure) — 32-bit, immediate offset	FEAT_AdvSIMD

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

L	R	Rm	opcode	S	size	Instruction Details	Feature
0	0	11111	101	0	01	ST3 (single structure) — 64-bit, immediate offset	FEAT_AdvSIMD
0	0	!= 11111	000			ST1 (single structure) — 8-bit, register offset	FEAT_AdvSIMD
0	0	!= 11111	001			ST3 (single structure) — 8-bit, register offset	FEAT_AdvSIMD
0	0	!= 11111	010		x0	ST1 (single structure) — 16-bit, register offset	FEAT_AdvSIMD
0	0	!= 11111	011		x0	ST3 (single structure) — 16-bit, register offset	FEAT_AdvSIMD
0	0	!= 11111	100		00	ST1 (single structure) — 32-bit, register offset	FEAT_AdvSIMD
0	0	!= 11111	100	0	01	ST1 (single structure) — 64-bit, register offset	FEAT_AdvSIMD
0	0	!= 11111	101		00	ST3 (single structure) — 32-bit, register offset	FEAT_AdvSIMD
0	0	!= 11111	101	0	01	ST3 (single structure) — 64-bit, register offset	FEAT_AdvSIMD
0	1	11111	000			ST2 (single structure) — 8-bit, immediate offset	FEAT_AdvSIMD
0	1	11111	001			ST4 (single structure) — 8-bit, immediate offset	FEAT_AdvSIMD
0	1	11111	010		x0	ST2 (single structure) — 16-bit, immediate offset	FEAT_AdvSIMD
0	1	11111	011		x0	ST4 (single structure) — 16-bit, immediate offset	FEAT_AdvSIMD
0	1	11111	100		00	ST2 (single structure) — 32-bit, immediate offset	FEAT_AdvSIMD
0	1	11111	100	0	01	ST2 (single structure) — 64-bit, immediate offset	FEAT_AdvSIMD
0	1	11111	101		00	ST4 (single structure) — 32-bit, immediate offset	FEAT_AdvSIMD
0	1	11111	101	0	01	ST4 (single structure) — 64-bit, immediate offset	FEAT_AdvSIMD
0	1	!= 11111	000			ST2 (single structure) — 8-bit, register offset	FEAT_AdvSIMD
0	1	!= 11111	001			ST4 (single structure) — 8-bit, register offset	FEAT_AdvSIMD
0	1	!= 11111	010		x0	ST2 (single structure) — 16-bit, register offset	FEAT_AdvSIMD
0	1	!= 11111	011		x0	ST4 (single structure) — 16-bit, register offset	FEAT_AdvSIMD
0	1	!= 11111	100		00	ST2 (single structure) — 32-bit, register offset	FEAT_AdvSIMD
0	1	!= 11111	100	0	01	ST2 (single structure) — 64-bit, register offset	FEAT_AdvSIMD
0	1	!= 11111	101		00	ST4 (single structure) — 32-bit, register offset	FEAT_AdvSIMD
0	1	!= 11111	101	0	01	ST4 (single structure) — 64-bit, register offset	FEAT_AdvSIMD
1			11x	1		UNALLOCATED	-

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

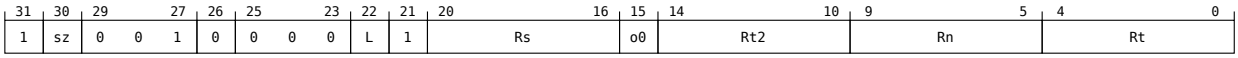
L	R	Rm	opcode	S	size	Instruction Details	Feature
1	0	11111	000			LD1 (single structure) — 8-bit, immediate offset	FEAT_AdvSIMD
1	0	11111	001			LD3 (single structure) — 8-bit, immediate offset	FEAT_AdvSIMD
1	0	11111	010		x0	LD1 (single structure) — 16-bit, immediate offset	FEAT_AdvSIMD
1	0	11111	011		x0	LD3 (single structure) — 16-bit, immediate offset	FEAT_AdvSIMD
1	0	11111	100		00	LD1 (single structure) — 32-bit, immediate offset	FEAT_AdvSIMD
1	0	11111	100	0	01	LD1 (single structure) — 64-bit, immediate offset	FEAT_AdvSIMD
1	0	11111	101		00	LD3 (single structure) — 32-bit, immediate offset	FEAT_AdvSIMD
1	0	11111	101	0	01	LD3 (single structure) — 64-bit, immediate offset	FEAT_AdvSIMD
1	0	11111	110	0		LD1R — Immediate offset	FEAT_AdvSIMD
1	0	11111	111	0		LD3R — Immediate offset	FEAT_AdvSIMD
1	0	!= 11111	000			LD1 (single structure) — 8-bit, register offset	FEAT_AdvSIMD
1	0	!= 11111	001			LD3 (single structure) — 8-bit, register offset	FEAT_AdvSIMD
1	0	!= 11111	010		x0	LD1 (single structure) — 16-bit, register offset	FEAT_AdvSIMD
1	0	!= 11111	011		x0	LD3 (single structure) — 16-bit, register offset	FEAT_AdvSIMD
1	0	!= 11111	100		00	LD1 (single structure) — 32-bit, register offset	FEAT_AdvSIMD
1	0	!= 11111	100	0	01	LD1 (single structure) — 64-bit, register offset	FEAT_AdvSIMD
1	0	!= 11111	101		00	LD3 (single structure) — 32-bit, register offset	FEAT_AdvSIMD
1	0	!= 11111	101	0	01	LD3 (single structure) — 64-bit, register offset	FEAT_AdvSIMD
1	0	!= 11111	110	0		LD1R — Register offset	FEAT_AdvSIMD
1	0	!= 11111	111	0		LD3R — Register offset	FEAT_AdvSIMD
1	1	11111	000			LD2 (single structure) — 8-bit, immediate offset	FEAT_AdvSIMD
1	1	11111	001			LD4 (single structure) — 8-bit, immediate offset	FEAT_AdvSIMD
1	1	11111	010		x0	LD2 (single structure) — 16-bit, immediate offset	FEAT_AdvSIMD
1	1	11111	011		x0	LD4 (single structure) — 16-bit, immediate offset	FEAT_AdvSIMD
1	1	11111	100		00	LD2 (single structure) — 32-bit, immediate offset	FEAT_AdvSIMD
1	1	11111	100	0	01	LD2 (single structure) — 64-bit, immediate offset	FEAT_AdvSIMD
1	1	11111	101		00	LD4 (single structure) — 32-bit, immediate offset	FEAT_AdvSIMD

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

L	R	Rm	opcode	S	size	Instruction Details	Feature
1	1	11111	101	0	01	LD4 (single structure) — 64-bit, immediate offset	FEAT_AdvSIMD
1	1	11111	110	0		LD2R — Immediate offset	FEAT_AdvSIMD
1	1	11111	111	0		LD4R — Immediate offset	FEAT_AdvSIMD
1	1	!= 11111	000			LD2 (single structure) — 8-bit, register offset	FEAT_AdvSIMD
1	1	!= 11111	001			LD4 (single structure) — 8-bit, register offset	FEAT_AdvSIMD
1	1	!= 11111	010		x0	LD2 (single structure) — 16-bit, register offset	FEAT_AdvSIMD
1	1	!= 11111	011		x0	LD4 (single structure) — 16-bit, register offset	FEAT_AdvSIMD
1	1	!= 11111	100		00	LD2 (single structure) — 32-bit, register offset	FEAT_AdvSIMD
1	1	!= 11111	100	0	01	LD2 (single structure) — 64-bit, register offset	FEAT_AdvSIMD
1	1	!= 11111	101		00	LD4 (single structure) — 32-bit, register offset	FEAT_AdvSIMD
1	1	!= 11111	101	0	01	LD4 (single structure) — 64-bit, register offset	FEAT_AdvSIMD
1	1	!= 11111	110	0		LD2R — Register offset	FEAT_AdvSIMD
1	1	!= 11111	111	0		LD4R — Register offset	FEAT_AdvSIMD

C4.1.6.6 Load/store exclusive pair

The encodings in this section are decoded from Loads and Stores.

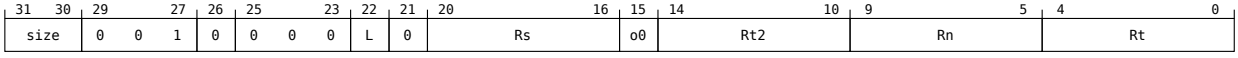


sz	L	o0	Instruction Details
0	0	0	STXP — 32-bit
0	0	1	STLXP — 32-bit
0	1	0	LDXP — 32-bit
0	1	1	LDAXP — 32-bit
1	0	0	STXP — 64-bit
1	0	1	STLXP — 64-bit
1	1	0	LDXP — 64-bit
1	1	1	LDAXP — 64-bit

C4.1.6.7 Load/store exclusive register

The encodings in this section are decoded from Loads and Stores.

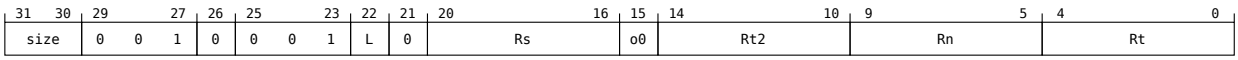
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential



size	L	o0	Instruction Details
00	0	0	STXRB
00	0	1	STLXRB
00	1	0	LDXRB
00	1	1	LDAXRB
01	0	0	STXRH
01	0	1	STLXRH
01	1	0	LDXRH
01	1	1	LDAXRH
10	0	0	STXR — 32-bit
10	0	1	STLXR — 32-bit
10	1	0	LDXR — 32-bit
10	1	1	LDAXR — 32-bit
11	0	0	STXR — 64-bit
11	0	1	STLXR — 64-bit
11	1	0	LDXR — 64-bit
11	1	1	LDAXR — 64-bit

C4.1.6.8 Load/store ordered

The encodings in this section are decoded from Loads and Stores.

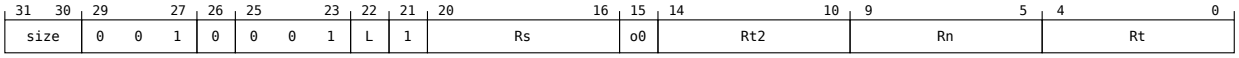


size	L	o0	Instruction Details
		0	UNALLOCATED
00	0	1	STLRB
00	1	1	LDARB
01	0	1	STLRH
01	1	1	LDARH
10	0	1	STLR — 32-bit
10	1	1	LDAR — 32-bit
11	0	1	STLR — 64-bit
11	1	1	LDAR — 64-bit

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C4.1.6.9 Compare and swap

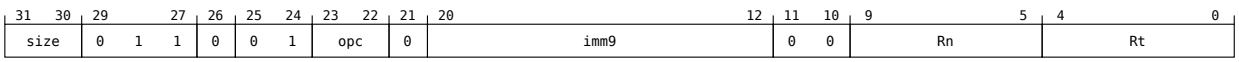
The encodings in this section are decoded from Loads and Stores.



size	L	o0	Rt2	Instruction Details	Feature
			!= 11111	UNALLOCATED	-
00	0	0	11111	CASB, CASAB, CASALB, CASLB — CASB	FEAT_LSE
00	0	1	11111	CASB, CASAB, CASALB, CASLB — CASLB	FEAT_LSE
00	1	0	11111	CASB, CASAB, CASALB, CASLB — CASAB	FEAT_LSE
00	1	1	11111	CASB, CASAB, CASALB, CASLB — CASALB	FEAT_LSE
01	0	0	11111	CASH, CASAH, CASALH, CASLH — CASH	FEAT_LSE
01	0	1	11111	CASH, CASAH, CASALH, CASLH — CASLH	FEAT_LSE
01	1	0	11111	CASH, CASAH, CASALH, CASLH — CASAH	FEAT_LSE
01	1	1	11111	CASH, CASAH, CASALH, CASLH — CASALH	FEAT_LSE
10	0	0	11111	CAS, CASA, CASAL, CASL — 32-bit CAS	FEAT_LSE
10	0	1	11111	CAS, CASA, CASAL, CASL — 32-bit CASL	FEAT_LSE
10	1	0	11111	CAS, CASA, CASAL, CASL — 32-bit CASA	FEAT_LSE
10	1	1	11111	CAS, CASA, CASAL, CASL — 32-bit CASAL	FEAT_LSE
11	0	0	11111	CAS, CASA, CASAL, CASL — 64-bit CAS	FEAT_LSE
11	0	1	11111	CAS, CASA, CASAL, CASL — 64-bit CASL	FEAT_LSE
11	1	0	11111	CAS, CASA, CASAL, CASL — 64-bit CASA	FEAT_LSE
11	1	1	11111	CAS, CASA, CASAL, CASL — 64-bit CASAL	FEAT_LSE

C4.1.6.10 Load/store ordered (unscaled immediate)

The encodings in this section are decoded from Loads and Stores.

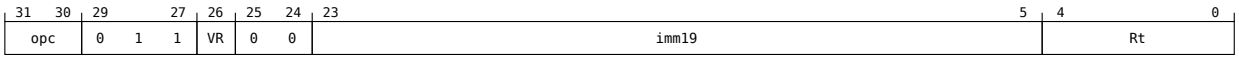


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	opc	Instruction Details	Feature
00	00	STLURB	FEAT_LRCPC2
00	01	LDAPURB	FEAT_LRCPC2
00	10	LDAPURSB — 64-bit	FEAT_LRCPC2
00	11	LDAPURSB — 32-bit	FEAT_LRCPC2
01	00	STLURH	FEAT_LRCPC2
01	01	LDAPURH	FEAT_LRCPC2
01	10	LDAPURSH — 64-bit	FEAT_LRCPC2
01	11	LDAPURSH — 32-bit	FEAT_LRCPC2
10	00	STLUR — 32-bit	FEAT_LRCPC2
10	01	LDAPUR — 32-bit	FEAT_LRCPC2
10	10	LDAPURSW	FEAT_LRCPC2
10	11	UNALLOCATED	-
11	00	STLUR — 64-bit	FEAT_LRCPC2
11	01	LDAPUR — 64-bit	FEAT_LRCPC2
11	1x	UNALLOCATED	-

C4.1.6.11 Load register (literal)

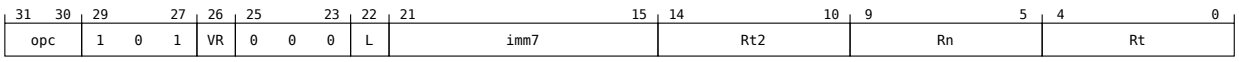
The encodings in this section are decoded from Loads and Stores.



opc	VR	Instruction Details	Feature
00	0	LDR (literal) — 32-bit	-
00	1	LDR (literal, SIMD&FP) — 32-bit	FEAT_FP
01	0	LDR (literal) — 64-bit	-
01	1	LDR (literal, SIMD&FP) — 64-bit	FEAT_FP
10	0	LDRSW (literal)	-
10	1	LDR (literal, SIMD&FP) — 128-bit	FEAT_FP
11	0	PRFM (literal)	-
11	1	UNALLOCATED	-

C4.1.6.12 Load/store no-allocate pair (offset)

The encodings in this section are decoded from Loads and Stores.

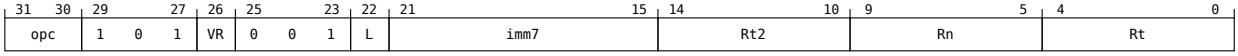


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

opc	VR	L	Instruction Details	Feature
00	0	0	STNP — 32-bit	-
00	0	1	LDNP — 32-bit	-
00	1	0	STNP (SIMD&FP) — 32-bit	FEAT_FP
00	1	1	LDNP (SIMD&FP) — 32-bit	FEAT_FP
01	0		UNALLOCATED	-
01	1	0	STNP (SIMD&FP) — 64-bit	FEAT_FP
01	1	1	LDNP (SIMD&FP) — 64-bit	FEAT_FP
10	0	0	STNP — 64-bit	-
10	0	1	LDNP — 64-bit	-
10	1	0	STNP (SIMD&FP) — 128-bit	FEAT_FP
10	1	1	LDNP (SIMD&FP) — 128-bit	FEAT_FP
11			UNALLOCATED	-

C4.1.6.13 Load/store register pair (post-indexed)

The encodings in this section are decoded from Loads and Stores.

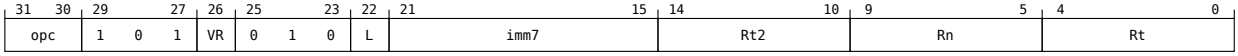


opc	VR	L	Instruction Details	Feature
00	0	0	STP — 32-bit	-
00	0	1	LDP — 32-bit	-
00	1	0	STP (SIMD&FP) — 32-bit	FEAT_FP
00	1	1	LDP (SIMD&FP) — 32-bit	FEAT_FP
01	0	0	UNALLOCATED	-
01	0	1	LDPSW	-
01	1	0	STP (SIMD&FP) — 64-bit	FEAT_FP
01	1	1	LDP (SIMD&FP) — 64-bit	FEAT_FP
10	0	0	STP — 64-bit	-
10	0	1	LDP — 64-bit	-
10	1	0	STP (SIMD&FP) — 128-bit	FEAT_FP
10	1	1	LDP (SIMD&FP) — 128-bit	FEAT_FP
11			UNALLOCATED	-

C4.1.6.14 Load/store register pair (offset)

The encodings in this section are decoded from Loads and Stores.

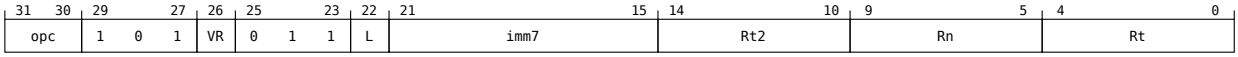
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential



opc	VR	L	Instruction Details	Feature
00	0	0	STP — 32-bit	-
00	0	1	LDP — 32-bit	-
00	1	0	STP (SIMD&FP) — 32-bit	FEAT_FP
00	1	1	LDP (SIMD&FP) — 32-bit	FEAT_FP
01	0	0	UNALLOCATED	-
01	0	1	LDPSW	-
01	1	0	STP (SIMD&FP) — 64-bit	FEAT_FP
01	1	1	LDP (SIMD&FP) — 64-bit	FEAT_FP
10	0	0	STP — 64-bit	-
10	0	1	LDP — 64-bit	-
10	1	0	STP (SIMD&FP) — 128-bit	FEAT_FP
10	1	1	LDP (SIMD&FP) — 128-bit	FEAT_FP
11			UNALLOCATED	-

C4.1.6.15 Load/store register pair (pre-indexed)

The encodings in this section are decoded from Loads and Stores.

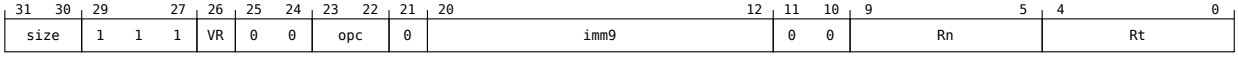


opc	VR	L	Instruction Details	Feature
00	0	0	STP — 32-bit	-
00	0	1	LDP — 32-bit	-
00	1	0	STP (SIMD&FP) — 32-bit	FEAT_FP
00	1	1	LDP (SIMD&FP) — 32-bit	FEAT_FP
01	0	0	UNALLOCATED	-
01	0	1	LDPSW	-
01	1	0	STP (SIMD&FP) — 64-bit	FEAT_FP
01	1	1	LDP (SIMD&FP) — 64-bit	FEAT_FP
10	0	0	STP — 64-bit	-
10	0	1	LDP — 64-bit	-
10	1	0	STP (SIMD&FP) — 128-bit	FEAT_FP
10	1	1	LDP (SIMD&FP) — 128-bit	FEAT_FP
11			UNALLOCATED	-

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C4.1.6.16 Load/store register (unscaled immediate)

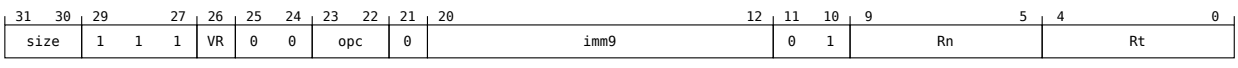
The encodings in this section are decoded from Loads and Stores.



size	VR	opc	Instruction Details	Feature
00	0	00	STURB	-
00	0	01	LDURB	-
00	0	10	LDURSB — 64-bit	-
00	0	11	LDURSB — 32-bit	-
00	1	00	STUR (SIMD&FP) — 8-bit	FEAT_FP
00	1	01	LDUR (SIMD&FP) — 8-bit	FEAT_FP
00	1	10	STUR (SIMD&FP) — 128-bit	FEAT_FP
00	1	11	LDUR (SIMD&FP) — 128-bit	FEAT_FP
01	0	00	STURH	-
01	0	01	LDURH	-
01	0	10	LDURSH — 64-bit	-
01	0	11	LDURSH — 32-bit	-
01	1	00	STUR (SIMD&FP) — 16-bit	FEAT_FP
01	1	01	LDUR (SIMD&FP) — 16-bit	FEAT_FP
1x	0	11	UNALLOCATED	-
10	0	00	STUR — 32-bit	-
10	0	01	LDUR — 32-bit	-
10	0	10	LDURSW	-
10	1	00	STUR (SIMD&FP) — 32-bit	FEAT_FP
10	1	01	LDUR (SIMD&FP) — 32-bit	FEAT_FP
11	0	00	STUR — 64-bit	-
11	0	01	LDUR — 64-bit	-
11	0	10	PRFUM	-
11	1	00	STUR (SIMD&FP) — 64-bit	FEAT_FP
11	1	01	LDUR (SIMD&FP) — 64-bit	FEAT_FP
!= 00	1	1x	UNALLOCATED	-

C4.1.6.17 Load/store register (immediate post-indexed)

The encodings in this section are decoded from Loads and Stores.

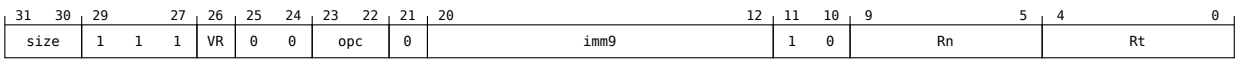


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	opc	Instruction Details	Feature
00	0	00	STRB (immediate)	-
00	0	01	LDRB (immediate)	-
00	0	10	LDRSB (immediate) — 64-bit	-
00	0	11	LDRSB (immediate) — 32-bit	-
00	1	00	STR (immediate, SIMD&FP) — 8-bit	FEAT_FP
00	1	01	LDR (immediate, SIMD&FP) — 8-bit	FEAT_FP
00	1	10	STR (immediate, SIMD&FP) — 128-bit	FEAT_FP
00	1	11	LDR (immediate, SIMD&FP) — 128-bit	FEAT_FP
01	0	00	STRH (immediate)	-
01	0	01	LDRH (immediate)	-
01	0	10	LDRSH (immediate) — 64-bit	-
01	0	11	LDRSH (immediate) — 32-bit	-
01	1	00	STR (immediate, SIMD&FP) — 16-bit	FEAT_FP
01	1	01	LDR (immediate, SIMD&FP) — 16-bit	FEAT_FP
01	1	1x	UNALLOCATED	-
10	0	00	STR (immediate) — 32-bit	-
10	0	01	LDR (immediate) — 32-bit	-
10	0	10	LDRSW (immediate)	-
10	0	11	UNALLOCATED	-
10	1	00	STR (immediate, SIMD&FP) — 32-bit	FEAT_FP
10	1	01	LDR (immediate, SIMD&FP) — 32-bit	FEAT_FP
10	1	1x	UNALLOCATED	-
11		1x	UNALLOCATED	-
11	0	00	STR (immediate) — 64-bit	-
11	0	01	LDR (immediate) — 64-bit	-
11	1	00	STR (immediate, SIMD&FP) — 64-bit	FEAT_FP
11	1	01	LDR (immediate, SIMD&FP) — 64-bit	FEAT_FP

C4.1.6.18 Load/store register (unprivileged)

The encodings in this section are decoded from Loads and Stores.



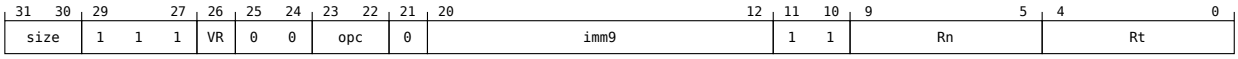
size	VR	opc	Instruction Details
	1		UNALLOCATED
00	0	00	STTRB
00	0	01	LDTRB

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	opc	Instruction Details
00	0	10	LDTRSB — 64-bit
00	0	11	LDTRSB — 32-bit
01	0	00	STTRH
01	0	01	LDTRH
01	0	10	LDTRSH — 64-bit
01	0	11	LDTRSH — 32-bit
10	0	00	STTR — 32-bit
10	0	01	LDTR — 32-bit
10	0	10	LDTRSW
10	0	11	UNALLOCATED
11	0	00	STTR — 64-bit
11	0	01	LDTR — 64-bit
11	0	1x	UNALLOCATED

C4.1.6.19 Load/store register (immediate pre-indexed)

The encodings in this section are decoded from Loads and Stores.



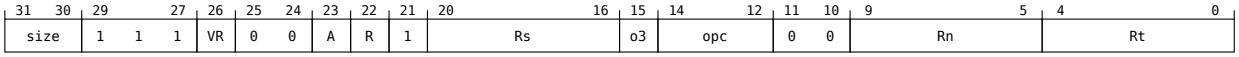
size	VR	opc	Instruction Details	Feature
00	0	00	STRB (immediate)	-
00	0	01	LDRB (immediate)	-
00	0	10	LDRSB (immediate) — 64-bit	-
00	0	11	LDRSB (immediate) — 32-bit	-
00	1	00	STR (immediate, SIMD&FP) — 8-bit	FEAT_FP
00	1	01	LDR (immediate, SIMD&FP) — 8-bit	FEAT_FP
00	1	10	STR (immediate, SIMD&FP) — 128-bit	FEAT_FP
00	1	11	LDR (immediate, SIMD&FP) — 128-bit	FEAT_FP
01	0	00	STRH (immediate)	-
01	0	01	LDRH (immediate)	-
01	0	10	LDRSH (immediate) — 64-bit	-
01	0	11	LDRSH (immediate) — 32-bit	-
01	1	00	STR (immediate, SIMD&FP) — 16-bit	FEAT_FP
01	1	01	LDR (immediate, SIMD&FP) — 16-bit	FEAT_FP
01	1	1x	UNALLOCATED	-
10	0	00	STR (immediate) — 32-bit	-
10	0	01	LDR (immediate) — 32-bit	-

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	opc	Instruction Details	Feature
10	0	10	LDRSW (immediate)	-
10	0	11	UNALLOCATED	-
10	1	00	STR (immediate, SIMD&FP) — 32-bit	FEAT_FP
10	1	01	LDR (immediate, SIMD&FP) — 32-bit	FEAT_FP
10	1	1x	UNALLOCATED	-
11		1x	UNALLOCATED	-
11	0	00	STR (immediate) — 64-bit	-
11	0	01	LDR (immediate) — 64-bit	-
11	1	00	STR (immediate, SIMD&FP) — 64-bit	FEAT_FP
11	1	01	LDR (immediate, SIMD&FP) — 64-bit	FEAT_FP

C4.1.6.20 Atomic memory operations

The encodings in this section are decoded from Loads and Stores.



size	VR	A	R	o3	opc	Instruction Details	Feature
	0			1	x01	UNALLOCATED	-
	0			1	x1x	UNALLOCATED	-
	0	0		1	100	UNALLOCATED	-
	0	1	1	1	100	UNALLOCATED	-
	1					UNALLOCATED	-
00	0	0	0	0	000	LDADDB, LDADDAB, LDADDALB, LDADDLB — No memory ordering	FEAT_LSE
00	0	0	0	0	001	LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB — No memory ordering	FEAT_LSE
00	0	0	0	0	010	LDEORB, LDEORAB, LDEORALB, LDEORLB — No memory ordering	FEAT_LSE
00	0	0	0	0	011	LDSETB, LDSETAB, LDSETALB, LDSETLB — No memory ordering	FEAT_LSE
00	0	0	0	0	100	LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB — No memory ordering	FEAT_LSE
00	0	0	0	0	101	LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB — No memory ordering	FEAT_LSE
00	0	0	0	0	110	LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB — No memory ordering	FEAT_LSE
00	0	0	0	0	111	LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB — No memory ordering	FEAT_LSE

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	A	R	o3	opc	Instruction Details	Feature
00	0	0	0	1	000	SWPB, SWPAB, SWPALB, SWPLB — SWPB	FEAT_LSE
00	0	0	1	0	000	LDADDB, LDADDAB, LDADDALB, LDADDLB — Release	FEAT_LSE
00	0	0	1	0	001	LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB — Release	FEAT_LSE
00	0	0	1	0	010	LDEORB, LDEORAB, LDEORALB, LDEORLB — Release	FEAT_LSE
00	0	0	1	0	011	LDSETB, LDSETAB, LDSETALB, LDSETLB — Release	FEAT_LSE
00	0	0	1	0	100	LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB — Release	FEAT_LSE
00	0	0	1	0	101	LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB — Release	FEAT_LSE
00	0	0	1	0	110	LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB — Release	FEAT_LSE
00	0	0	1	0	111	LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB — Release	FEAT_LSE
00	0	0	1	1	000	SWPB, SWPAB, SWPALB, SWPLB — SWPLB	FEAT_LSE
00	0	1	0	0	000	LDADDB, LDADDAB, LDADDALB, LDADDLB — Acquire	FEAT_LSE
00	0	1	0	0	001	LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB — Acquire	FEAT_LSE
00	0	1	0	0	010	LDEORB, LDEORAB, LDEORALB, LDEORLB — Acquire	FEAT_LSE
00	0	1	0	0	011	LDSETB, LDSETAB, LDSETALB, LDSETLB — Acquire	FEAT_LSE
00	0	1	0	0	100	LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB — Acquire	FEAT_LSE
00	0	1	0	0	101	LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB — Acquire	FEAT_LSE
00	0	1	0	0	110	LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB — Acquire	FEAT_LSE
00	0	1	0	0	111	LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB — Acquire	FEAT_LSE
00	0	1	0	1	000	SWPB, SWPAB, SWPALB, SWPLB — SWPAB	FEAT_LSE
00	0	1	0	1	100	LDAPRB	FEAT_LRCPC
00	0	1	1	0	000	LDADDB, LDADDAB, LDADDALB, LDADDLB — Acquire-release	FEAT_LSE
00	0	1	1	0	001	LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB — Acquire-release	FEAT_LSE
00	0	1	1	0	010	LDEORB, LDEORAB, LDEORALB, LDEORLB — Acquire-release	FEAT_LSE
00	0	1	1	0	011	LDSETB, LDSETAB, LDSETALB, LDSETLB — Acquire-release	FEAT_LSE

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	A	R	o3	opc	Instruction Details	Feature
00	0	1	1	0	100	LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB — Acquire-release	FEAT_LSE
00	0	1	1	0	101	LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB — Acquire-release	FEAT_LSE
00	0	1	1	0	110	LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB — Acquire-release	FEAT_LSE
00	0	1	1	0	111	LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB — Acquire-release	FEAT_LSE
00	0	1	1	1	000	SWPB, SWPAB, SWPALB, SWPLB — SWPALB	FEAT_LSE
01	0	0	0	0	000	LDADDH, LDADDAH, LDADDALH, LDADDLH — No memory ordering	FEAT_LSE
01	0	0	0	0	001	LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH — No memory ordering	FEAT_LSE
01	0	0	0	0	010	LDEORH, LDEORAH, LDEORALH, LDEORLH — No memory ordering	FEAT_LSE
01	0	0	0	0	011	LDSETH, LDSETAH, LDSETALH, LDSETLH — No memory ordering	FEAT_LSE
01	0	0	0	0	100	LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH — No memory ordering	FEAT_LSE
01	0	0	0	0	101	LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH — No memory ordering	FEAT_LSE
01	0	0	0	0	110	LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH — No memory ordering	FEAT_LSE
01	0	0	0	0	111	LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH — No memory ordering	FEAT_LSE
01	0	0	0	1	000	SWPH, SWPAH, SWPALH, SWPLH — SWPH	FEAT_LSE
01	0	0	1	0	000	LDADDH, LDADDAH, LDADDALH, LDADDLH — Release	FEAT_LSE
01	0	0	1	0	001	LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH — Release	FEAT_LSE
01	0	0	1	0	010	LDEORH, LDEORAH, LDEORALH, LDEORLH — Release	FEAT_LSE
01	0	0	1	0	011	LDSETH, LDSETAH, LDSETALH, LDSETLH — Release	FEAT_LSE
01	0	0	1	0	100	LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH — Release	FEAT_LSE
01	0	0	1	0	101	LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH — Release	FEAT_LSE
01	0	0	1	0	110	LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH — Release	FEAT_LSE
01	0	0	1	0	111	LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH — Release	FEAT_LSE

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	A	R	o3	opc	Instruction Details	Feature
01	0	0	1	1	000	SWPH, SWPAH, SWPALH, SWPLH — SWPLH	FEAT_LSE
01	0	1	0	0	000	LDADDH, LDADDAH, LDADDALH, LDADDLH — Acquire	FEAT_LSE
01	0	1	0	0	001	LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH — Acquire	FEAT_LSE
01	0	1	0	0	010	LDEORH, LDEORAH, LDEORALH, LDEORLH — Acquire	FEAT_LSE
01	0	1	0	0	011	LDSETH, LDSETAH, LDSETALH, LDSETLH — Acquire	FEAT_LSE
01	0	1	0	0	100	LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH — Acquire	FEAT_LSE
01	0	1	0	0	101	LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH — Acquire	FEAT_LSE
01	0	1	0	0	110	LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH — Acquire	FEAT_LSE
01	0	1	0	0	111	LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH — Acquire	FEAT_LSE
01	0	1	0	1	000	SWPH, SWPAH, SWPALH, SWPLH — SWPAH	FEAT_LSE
01	0	1	0	1	100	LDAPRH	FEAT_LRCPC
01	0	1	1	0	000	LDADDH, LDADDAH, LDADDALH, LDADDLH — Acquire-release	FEAT_LSE
01	0	1	1	0	001	LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH — Acquire-release	FEAT_LSE
01	0	1	1	0	010	LDEORH, LDEORAH, LDEORALH, LDEORLH — Acquire-release	FEAT_LSE
01	0	1	1	0	011	LDSETH, LDSETAH, LDSETALH, LDSETLH — Acquire-release	FEAT_LSE
01	0	1	1	0	100	LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH — Acquire-release	FEAT_LSE
01	0	1	1	0	101	LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH — Acquire-release	FEAT_LSE
01	0	1	1	0	110	LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH — Acquire-release	FEAT_LSE
01	0	1	1	0	111	LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH — Acquire-release	FEAT_LSE
01	0	1	1	1	000	SWPH, SWPAH, SWPALH, SWPLH — SWPALH	FEAT_LSE
10	0	0	0	0	000	LDADD, LDADDA, LDADDAL, LDADDL — 32-bit no memory ordering	FEAT_LSE
10	0	0	0	0	001	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 32-bit no memory ordering	FEAT_LSE
10	0	0	0	0	010	LDEOR, LDEORA, LDEORAL, LDEORL — 32-bit no memory ordering	FEAT_LSE

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	A	R	o3	opc	Instruction Details	Feature
10	0	0	0	0	011	LDSET, LDSETA, LDSETAL, LDSETL — 32-bit no memory ordering	FEAT_LSE
10	0	0	0	0	100	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 32-bit no memory ordering	FEAT_LSE
10	0	0	0	0	101	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 32-bit no memory ordering	FEAT_LSE
10	0	0	0	0	110	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 32-bit no memory ordering	FEAT_LSE
10	0	0	0	0	111	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 32-bit no memory ordering	FEAT_LSE
10	0	0	0	1	000	SWP, SWPA, SWPAL, SWPL — 32-bit SWP	FEAT_LSE
10	0	0	1	0	000	LDADD, LDADDA, LDADDAL, LDADDL — 32-bit release	FEAT_LSE
10	0	0	1	0	001	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 32-bit release	FEAT_LSE
10	0	0	1	0	010	LDEOR, LDEORA, LDEORAL, LDEORL — 32-bit release	FEAT_LSE
10	0	0	1	0	011	LDSET, LDSETA, LDSETAL, LDSETL — 32-bit release	FEAT_LSE
10	0	0	1	0	100	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 32-bit release	FEAT_LSE
10	0	0	1	0	101	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 32-bit release	FEAT_LSE
10	0	0	1	0	110	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 32-bit release	FEAT_LSE
10	0	0	1	0	111	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 32-bit release	FEAT_LSE
10	0	0	1	1	000	SWP, SWPA, SWPAL, SWPL — 32-bit SWPL	FEAT_LSE
10	0	1	0	0	000	LDADD, LDADDA, LDADDAL, LDADDL — 32-bit acquire	FEAT_LSE
10	0	1	0	0	001	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 32-bit acquire	FEAT_LSE
10	0	1	0	0	010	LDEOR, LDEORA, LDEORAL, LDEORL — 32-bit acquire	FEAT_LSE
10	0	1	0	0	011	LDSET, LDSETA, LDSETAL, LDSETL — 32-bit acquire	FEAT_LSE
10	0	1	0	0	100	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 32-bit acquire	FEAT_LSE
10	0	1	0	0	101	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 32-bit acquire	FEAT_LSE
10	0	1	0	0	110	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 32-bit acquire	FEAT_LSE
10	0	1	0	0	111	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 32-bit acquire	FEAT_LSE
10	0	1	0	1	000	SWP, SWPA, SWPAL, SWPL — 32-bit SWPA	FEAT_LSE
10	0	1	0	1	100	LDAPR — 32-bit	FEAT_LRCPC

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	A	R	o3	opc	Instruction Details	Feature
10	0	1	1	0	000	LDADD, LDADDA, LDADDAL, LDADDL — 32-bit acquire-release	FEAT_LSE
10	0	1	1	0	001	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 32-bit acquire-release	FEAT_LSE
10	0	1	1	0	010	LDEOR, LDEORA, LDEORAL, LDEORL — 32-bit acquire-release	FEAT_LSE
10	0	1	1	0	011	LDSET, LDSETA, LDSETAL, LDSETL — 32-bit acquire-release	FEAT_LSE
10	0	1	1	0	100	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 32-bit acquire-release	FEAT_LSE
10	0	1	1	0	101	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 32-bit acquire-release	FEAT_LSE
10	0	1	1	0	110	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 32-bit acquire-release	FEAT_LSE
10	0	1	1	0	111	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 32-bit acquire-release	FEAT_LSE
10	0	1	1	1	000	SWP, SWPA, SWPAL, SWPL — 32-bit SWPAL	FEAT_LSE
11	0	0	0	0	000	LDADD, LDADDA, LDADDAL, LDADDL — 64-bit no memory ordering	FEAT_LSE
11	0	0	0	0	001	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 64-bit no memory ordering	FEAT_LSE
11	0	0	0	0	010	LDEOR, LDEORA, LDEORAL, LDEORL — 64-bit no memory ordering	FEAT_LSE
11	0	0	0	0	011	LDSET, LDSETA, LDSETAL, LDSETL — 64-bit no memory ordering	FEAT_LSE
11	0	0	0	0	100	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 64-bit no memory ordering	FEAT_LSE
11	0	0	0	0	101	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 64-bit no memory ordering	FEAT_LSE
11	0	0	0	0	110	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 64-bit no memory ordering	FEAT_LSE
11	0	0	0	0	111	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 64-bit no memory ordering	FEAT_LSE
11	0	0	0	1	000	SWP, SWPA, SWPAL, SWPL — 64-bit SWP	FEAT_LSE
11	0	0	1	0	000	LDADD, LDADDA, LDADDAL, LDADDL — 64-bit release	FEAT_LSE
11	0	0	1	0	001	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 64-bit release	FEAT_LSE
11	0	0	1	0	010	LDEOR, LDEORA, LDEORAL, LDEORL — 64-bit release	FEAT_LSE
11	0	0	1	0	011	LDSET, LDSETA, LDSETAL, LDSETL — 64-bit release	FEAT_LSE
11	0	0	1	0	100	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 64-bit release	FEAT_LSE

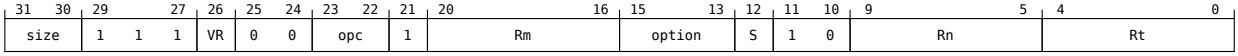
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	A	R	o3	opc	Instruction Details	Feature
11	0	0	1	0	101	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 64-bit release	FEAT_LSE
11	0	0	1	0	110	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 64-bit release	FEAT_LSE
11	0	0	1	0	111	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 64-bit release	FEAT_LSE
11	0	0	1	1	000	SWP, SWPA, SWPAL, SWPL — 64-bit SWPL	FEAT_LSE
11	0	1	0	0	000	LDADD, LDADDA, LDADDAL, LDADDL — 64-bit acquire	FEAT_LSE
11	0	1	0	0	001	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 64-bit acquire	FEAT_LSE
11	0	1	0	0	010	LDEOR, LDEORA, LDEORAL, LDEORL — 64-bit acquire	FEAT_LSE
11	0	1	0	0	011	LDSET, LDSETA, LDSETAL, LDSETL — 64-bit acquire	FEAT_LSE
11	0	1	0	0	100	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 64-bit acquire	FEAT_LSE
11	0	1	0	0	101	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 64-bit acquire	FEAT_LSE
11	0	1	0	0	110	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 64-bit acquire	FEAT_LSE
11	0	1	0	0	111	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 64-bit acquire	FEAT_LSE
11	0	1	0	1	000	SWP, SWPA, SWPAL, SWPL — 64-bit SWPA	FEAT_LSE
11	0	1	0	1	100	LDAPR — 64-bit	FEAT_LRCPC
11	0	1	1	0	000	LDADD, LDADDA, LDADDAL, LDADDL — 64-bit acquire-release	FEAT_LSE
11	0	1	1	0	001	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 64-bit acquire-release	FEAT_LSE
11	0	1	1	0	010	LDEOR, LDEORA, LDEORAL, LDEORL — 64-bit acquire-release	FEAT_LSE
11	0	1	1	0	011	LDSET, LDSETA, LDSETAL, LDSETL — 64-bit acquire-release	FEAT_LSE
11	0	1	1	0	100	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 64-bit acquire-release	FEAT_LSE
11	0	1	1	0	101	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 64-bit acquire-release	FEAT_LSE
11	0	1	1	0	110	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 64-bit acquire-release	FEAT_LSE
11	0	1	1	0	111	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 64-bit acquire-release	FEAT_LSE
11	0	1	1	1	000	SWP, SWPA, SWPAL, SWPL — 64-bit SWPAL	FEAT_LSE

C4.1.6.21 Load/store register (register offset)

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The encodings in this section are decoded from Loads and Stores.



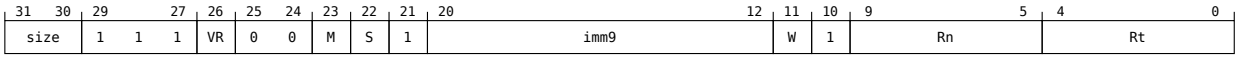
size	VR	opc	option	Rt	Instruction Details	Feature
00	0	00	011		STRB (register) — Shifted register	-
00	0	00	!= 011		STRB (register) — Extended register	-
00	0	01	011		LDRB (register) — Shifted register	-
00	0	01	!= 011		LDRB (register) — Extended register	-
00	0	10	011		LDRSB (register) — 64-bit with shifted register offset	-
00	0	10	!= 011		LDRSB (register) — 64-bit with extended register offset	-
00	0	11	011		LDRSB (register) — 32-bit with shifted register offset	-
00	0	11	!= 011		LDRSB (register) — 32-bit with extended register offset	-
00	1	00	011		STR (register, SIMD&FP) — 8-bit with shifted register offset	FEAT_FP
00	1	00	!= 011		STR (register, SIMD&FP) — 8-bit with extended register offset	FEAT_FP
00	1	01	011		LDR (register, SIMD&FP) — 8-bit with shifted register offset	FEAT_FP
00	1	01	!= 011		LDR (register, SIMD&FP) — 8-bit with extended register offset	FEAT_FP
00	1	10			STR (register, SIMD&FP) — 128-bit	FEAT_FP
00	1	11			LDR (register, SIMD&FP) — 128-bit	FEAT_FP
01	0	00			STRH (register)	-
01	0	01			LDRH (register)	-
01	0	10			LDRSH (register) — 64-bit	-
01	0	11			LDRSH (register) — 32-bit	-
01	1	00			STR (register, SIMD&FP) — 16-bit	FEAT_FP
01	1	01			LDR (register, SIMD&FP) — 16-bit	FEAT_FP
1x	0	11			UNALLOCATED	-
10	0	00			STR (register) — 32-bit	-
10	0	01			LDR (register) — 32-bit	-
10	0	10			LDRSW (register)	-
10	1	00			STR (register, SIMD&FP) — 32-bit	FEAT_FP
10	1	01			LDR (register, SIMD&FP) — 32-bit	FEAT_FP
11	0	00			STR (register) — 64-bit	-
11	0	01			LDR (register) — 64-bit	-
11	0	10	x0x		UNALLOCATED	-

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	opc	option	Rt	Instruction Details	Feature
11	0	10	x1x	11xxx	UNALLOCATED	-
11	0	10	x1x	!= 11xxx	PRFM (register)	-
11	1	00			STR (register, SIMD&FP) — 64-bit	FEAT_FP
11	1	01			LDR (register, SIMD&FP) — 64-bit	FEAT_FP
!= 00	1	1x			UNALLOCATED	-

C4.1.6.22 Load/store register (pac)

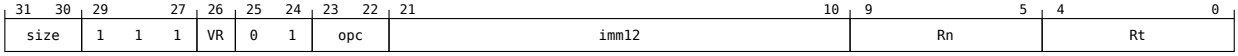
The encodings in this section are decoded from Loads and Stores.



size	VR	M	W	Instruction Details	Feature
11	0	0	0	LDRAA, LDRAB — Key A, offset	FEAT_PAuth
11	0	0	1	LDRAA, LDRAB — Key A, pre-indexed	FEAT_PAuth
11	0	1	0	LDRAA, LDRAB — Key B, offset	FEAT_PAuth
11	0	1	1	LDRAA, LDRAB — Key B, pre-indexed	FEAT_PAuth
11	1			UNALLOCATED	-
!= 11				UNALLOCATED	-

C4.1.6.23 Load/store register (unsigned immediate)

The encodings in this section are decoded from Loads and Stores.



size	VR	opc	Instruction Details	Feature
00	0	00	STRB (immediate)	-
00	0	01	LDRB (immediate)	-
00	0	10	LDRSB (immediate) — 64-bit	-
00	0	11	LDRSB (immediate) — 32-bit	-
00	1	00	STR (immediate, SIMD&FP) — 8-bit	FEAT_FP
00	1	01	LDR (immediate, SIMD&FP) — 8-bit	FEAT_FP
00	1	10	STR (immediate, SIMD&FP) — 128-bit	FEAT_FP
00	1	11	LDR (immediate, SIMD&FP) — 128-bit	FEAT_FP
01	0	00	STRH (immediate)	-
01	0	01	LDRH (immediate)	-

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	VR	opc	Instruction Details	Feature
01	0	10	LDRSH (immediate) — 64-bit	-
01	0	11	LDRSH (immediate) — 32-bit	-
01	1	00	STR (immediate, SIMD&FP) — 16-bit	FEAT_FP
01	1	01	LDR (immediate, SIMD&FP) — 16-bit	FEAT_FP
1x	0	11	UNALLOCATED	-
10	0	00	STR (immediate) — 32-bit	-
10	0	01	LDR (immediate) — 32-bit	-
10	0	10	LDRSW (immediate)	-
10	1	00	STR (immediate, SIMD&FP) — 32-bit	FEAT_FP
10	1	01	LDR (immediate, SIMD&FP) — 32-bit	FEAT_FP
11	0	00	STR (immediate) — 64-bit	-
11	0	01	LDR (immediate) — 64-bit	-
11	0	10	PRFM (immediate)	-
11	1	00	STR (immediate, SIMD&FP) — 64-bit	FEAT_FP
11	1	01	LDR (immediate, SIMD&FP) — 64-bit	FEAT_FP
!= 00	1	1x	UNALLOCATED	-

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter C5
The A64 System Instruction Class

This chapter describes the A64 System instruction class, and the System instruction class encoding space, that is a subset of the System registers encoding space. It contains the following sections:

- The System instruction class encoding space.
- Special purpose registers.
- A64 System instructions for cache maintenance.
- A64 System instructions for address translation.
- A64 System instructions for TLB maintenance.
- A64 System instructions for prediction restriction.
- A64 IMPLEMENTATION DEFINED System instructions.

See General information about the A64 instruction descriptions for information about entries used in the instruction encoding descriptions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.1 The System instruction class encoding space

Part of the A64 instruction encoding space is assigned to instructions that access the System register encoding space. These instructions provide:

- Access to *System registers*, including the debug registers, that provide system control, and system status information.
- Access to Special-purpose registers such as SPSR_ELx, ELR_ELx, and the equivalent fields of the Process State.
- The cache and TLB maintenance instructions and address translation instructions.
- Barriers and the `CLREX` instruction.
- Architectural hint instructions.

This section describes the general model for accessing this functionality.

Note

- See Fixed values in AArch64 instruction and System register descriptions for information about abbreviations used in the System instruction descriptions.

Principles of the System instruction class encoding describes some general properties of these encodings. System instruction class encoding overview then describes the top-level encoding of these instructions, and the following sections then describe the next level of the encoding hierarchy of System instructions and Special-purpose registers:

- op0 == `0b00`, architectural hints, barriers and CLREX, and PSTATE access.
- op0 == `0b01`, cache maintenance, TLB maintenance, address translation and prediction restriction instructions.
- op0 == `0b11`, Moves to and from Special-purpose registers.

For the description of the next level of encoding hierarchy of System registers, see:

- Moves to and from debug and trace System registers.
- Moves to and from non-debug System registers, Special-purpose registers.
- Reserved encodings for IMPLEMENTATION DEFINED registers.

C5.1.1 Principles of the System instruction class encoding

An encoding in the System instruction space is identified by a set of arguments,{`op0`, `op1`, `CRn`, `CRm`, `op2`}. These form an encoding hierarchy, where:

`op0` Defines the top-level division of the encoding space, see System instruction class encoding overview.

`op1` Identifies the lowest Exception level at which the encoding is accessible, as follows:

Accessible at EL0 `op1` has the value `0b011`.

Accessible at EL1 `op1` has the value 0, 1, or 2..

Accessible at Secure EL1 `op1` has the value `0b111`.

Accessible at EL2 `op1` has the value `0b100`.

Arm strongly recommends that implementers adopt this use of `op1` when using the IMPLEMENTATION DEFINED regions of the encoding space described in Reserved encodings for IMPLEMENTATION DEFINED registers.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

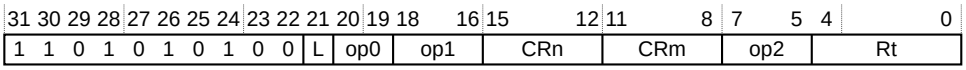
C5.1.2 System instruction class encoding overview

The encoding of the System instruction class describes each instruction as being either:

- A transfer to a System register. This is a System instruction with the semantics of a write.
- A transfer from a System register. This is a System instruction with the semantics of a read.

A System instruction that initiates an operation operates as if it was making a transfer to a register.

In the AArch64 instruction set, the decode structure for the System instruction class is:



The value of `L` indicates the transfer direction:

0 Transfer to System register.

1 Transfer from System register.

The `op0` field is the top-level encoding of the System instruction type. Its possible values are:

`0b00` These encodings provide:

- Instructions with an immediate field for accessing PSTATE, the current PE state.
- The architectural hint instructions.
- Barriers and the `CLREX` instruction.

For more information about these encodings, see op0 == `0b00`, architectural hints, barriers and CLREX, and PSTATE access.

`0b01` These encodings provide:

- Cache maintenance instructions.
- TLB maintenance instructions.
- Address translation instructions.
- Prediction restriction instructions.

For more information, see op0 == `0b01`, cache maintenance, TLB maintenance, address translation and prediction restriction instructions.

`0b10` These encodings provide moves to and from:

- Debug and trace registers.

For more information, see Moves to and from debug and trace System registers.

`0b11` These encodings provide:

- Moves to and from Non-debug System registers. The accessed registers provide system control, and system status information.
- Access to Special-purpose registers.

For more information, see Instructions for accessing Special-purpose registers and Instructions for accessing non-debug System registers.

C5.1.2.1 UNDEFINED behaviors

In the System register instruction encoding space, the following principles apply:

- All unallocated encodings are treated as UNDEFINED.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- All encodings with `L` == 1 and `op0` == `0b0x` are UNDEFINED, except for encodings in the area reserved for IMPLEMENTATION DEFINED use, see Reserved encoding space for IMPLEMENTATION DEFINED instructions.

For registers and operations that are accessible from a particular Exception level, any attempt to access those registers from a lower Exception level is UNDEFINED.

If a particular Exception level:

- Defines a register to be RO, then any attempt to write to that register, at that Exception level, is UNDEFINED. This means that any access to that register with `L` == 0 is UNDEFINED.
- Defines a register to be WO, then any attempt to read from that register, at that Exception level, is UNDEFINED. This means that any access to that register with `L` == 1 is UNDEFINED.

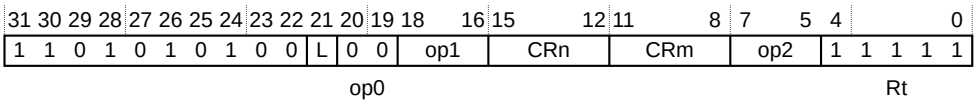
For IMPLEMENTATION DEFINED encoding spaces, the treatment of the encodings is IMPLEMENTATION DEFINED, but see the recommendation in Principles of the System instruction class encoding.

C5.1.3 `op0`== `0b00`, architectural hints, barriers and `CLREX`, and PSTATE access

The different groups of System register instructions with `op0`== `0b00`:

- Are identified by the value of `CRn`.
- Are always encoded with a value of `0b11111` in the `Rt` field.

The encoding of these instructions is:



The encoding of the `CRn` field is as follows:

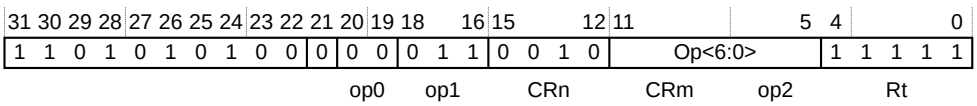
`0b0010` See Architectural hint instructions.

`0b0011` See Barriers and CLREX.

`0b0100` See Instructions for accessing the PSTATE fields.

C5.1.3.1 Architectural hint instructions

Within the `op0==0b00` encodings, the architectural hint instructions are identified by `CRn` having the value `0b0010`. The encoding of these instructions is:



The value of `op<6:0>`, formed by concatenating the `CRm` and `op2` fields, determines the hint instruction as follows:

`0b0000000` `NOP` instruction.

`0b0000001` `YIELD` instruction.

`0b0000010` `WFE` instruction.

`0b0000011` `WFI` instruction.

`0b0000100` `SEV` instruction.

`0b0000101` `SEVL` instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b0000110 DGH instruction.

0b0000111 XPACD, XPACI, XPACLRI instruction.

0b0001000 PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA instruction, PACIA1716 variant.

0b0001010 PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB instruction, PACIB1716 variant.

0b0001100 AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA instruction, AUTIA1716 variant.

0b0001110 AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB instruction, AUTIB1716 variant.

0b0010000 ESB instruction.

0b0010001

: TSB instruction.

0b0010100 CSDB instruction.

0b0011000 PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA instruction, PACIAZ variant.

0b0011001 PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA instruction, PACIASP variant.

0b0011010 PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB instruction, PACIBZ variant.

0b0011011 PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB instruction, PACIBSP variant.

0b0011100 AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA instruction, AUTIAZ variant.

0b0011101 AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA instruction, AUTIASP variant.

0b0011110 AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB instruction, AUTIBZ variant.

0b0011111 AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB instruction, AUTIBSP variant.

These instructions are described in A64 Base Instruction Descriptions.

Note

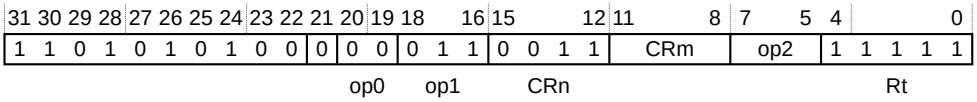
Instruction encodings with bits[4:0] not set to 0b11111 are UNDEFINED.

For more information about:

- The WFE, WFI, SEV, and SEVL instructions, see Mechanisms for entering a low-power state.
- The YIELD instruction, see Software control features and EL0.

C5.1.3.2 Barriers and CLREX

Within the op0==0b00 encodings, the barriers and CLREX instructions are identified by CRn having the value 0b0011. The encoding of these instructions is:



The value of op2 determines the instruction, as follows.

0b001 DSB instruction, Memory nXS barrier variant.

0b010 CLREX instruction.

0b100 DSB instruction, Memory barrier variant.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0b101` `DMB` instruction.

`0b110` `ISB` instruction.

`0b111` `SB` instruction.

`0b000`; `0b011` UNDEFINED.

These instructions are described in A64 Base Instruction Descriptions.

Note

- Instruction encodings with bits[4:0] not set to `0b11111` are UNDEFINED.

For more information about:

- The barrier instructions, see Memory barriers.
- The `CLREX` instruction, see Synchronization and semaphores.

C5.1.3.3 Instructions for accessing the PSTATE fields

Within the `op0==0b00` encodings, the instructions that can be used to modify PSTATE fields directly are identified by `CRn` having the value `0b0100`. The encoding of these instructions is shown in Figure C5-1 and Figure C5-2:

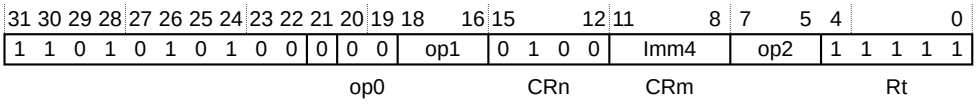


Figure C5-1 Instructions using `#Imm4`

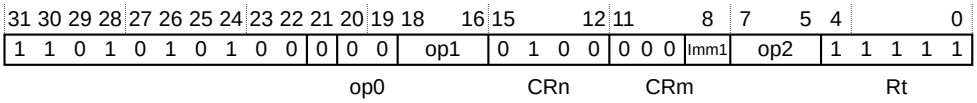


Figure C5-2 Instructions using `#Imm1`

These instructions are:

```
CFINV                ; Inverts the value of PSTATE.C
MSR DAIFSet, #Imm4   ; Used to set any or all of DAIF to 1
MSR DAIFClr, #Imm4   ; Used to clear any or all of DAIF to 0
MSR SPSel, #Imm4     ; Used to select the Stack Pointer, between SP_EL0 and SP_ELx
MSR UAO, #Imm4       ; Used to set the value of PSTATE.UAO
MSR PAN, #Imm4       ; Used to set the value of PSTATE.PAN
MSR DIT, #Imm4       ; Used to set the value of PSTATE.DIT
MSR SSBS, #Imm4      ; Used to set the value of PSTATE.SSBS
```

The value of `op2` selects the instruction form, which defines the constraints on the values of the `op1` and `Imm4` or `Imm1` arguments, as follows:

`op2` == `0b000` Selects the `CFINV` instruction.

`op2` == `0b011` Selects the `MSR UAO` instruction.

`op2` == `0b100` Selects the `MSR PAN` instruction.

`op2` == `0b101` Selects the `MSR SPSel` instruction.

`op2` == `0b001` Selects the `MSR SSBS` instruction.

`op2` == `0b010` Selects the `MSR DIT` instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`op2` == `0b100` Reserved.

`op2` == `0b110` Selects the `MSR DAIFSet` instruction, that sets the specified PSTATE.{D, A, I, F} bits to 1.

`op2` == `0b111` Selects the `MSR DAIFClr` instruction, that clears the specified PSTATE.{D, A, I, F} bits to 0.

`op2` == `0b000` Reserved.

All other combinations of `op1` and `op2` are reserved, and the corresponding instructions are UNDEFINED.

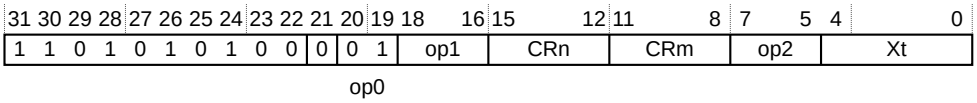
Note

For PSTATE updates, instruction encodings with bits[4:0] not set to `0b11111` are UNDEFINED.

Writes to PSTATE occur in program order without the need for additional synchronization. Changing PSTATE.SP to use SP_EL0 synchronizes any updates to SP_EL0 that have been written by an `MSR` to SP_EL0, without the need for additional synchronization.

C5.1.4 `op0`== `0b01`, cache maintenance, TLB maintenance, address translation and prediction restriction instructions

The System instructions are encoded with `op0`== `0b01`. The different groups of System instructions are identified by the values of `CRn` and `CRm`, except that some of this encoding space is reserved for IMPLEMENTATION DEFINED functionality. The encoding of these instructions is:



The grouping of these instructions depending on the `CRn` and `CRm` fields is as follows:

`CRn` == `0b0111` The instruction group is determined by the value of `CRm`, as follows:

`CRm` == {`0b0001`, `0b0101`} Instruction cache maintenance instructions.

See Cache maintenance instructions, and data cache zero operations.

`CRm` == `0b0011` Prediction restriction instructions.

See Prediction restriction instructions.

`CRm` == `0b0100` Data cache zero operations.

See Cache maintenance instructions, and data cache zero operations.

`CRm` == {`0b0110`, `0b1010`, `0b1011`, `0b1100`, `0b1101`, `0b1110`} Data cache maintenance instructions.

See Cache maintenance instructions, and data cache zero operations.

`CRm` == `0b1000` See Address translation instructions.

`CRn` == {`0b1000`, `0b1001`} See TLB maintenance instructions.

`CRn` == {`0b1011`, `0b1111`} See Reserved encoding space for IMPLEMENTATION DEFINED instructions.

C5.1.4.1 Cache maintenance instructions, and data cache zero operations

Table C5-1 lists the Cache maintenance instructions and their encodings. Instructions that take an argument include `Xt` in the instruction syntax. For instructions that do not take an argument, the Xt field is encoded as `0b11111`. For these instructions, if the Xt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Xt field is set to `0b11111`.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table C5-1 Cache maintenance instructions and data cache zero operations

op0	op1	CRn	CRm	op2	Mnemonic
0b01	0b000	0b0111	0b0001	0b000	IC IALLUIS
0b01	0b000	0b0111	0b0101	0b000	IC IALLU
0b01	0b000	0b0111	0b0110	0b001	DC IVAC
0b01	0b000	0b0111	0b0110	0b010	DC ISW
0b01	0b000	0b0111	0b1010	0b010	DC CSW
0b01	0b000	0b0111	0b1110	0b010	DC CISW
0b01	0b011	0b0111	0b0100	0b001	DC ZVA
0b01	0b011	0b0111	0b0101	0b001	IC IVAU
0b01	0b011	0b0111	0b1010	0b001	DC CVAC
0b01	0b011	0b0111	0b1011	0b001	DC CVAU
0b01	0b011	0b0111	0b1100	0b001	DC CVAP
0b01	0b011	0b0111	0b1101	0b001	DC CVADP
0b01	0b011	0b0111	0b1110	0b001	DC CIVAC

For more information about these instructions, see About cache maintenance in AArch64 state and A64 Cache maintenance instructions.

C5.1.4.2 Address translation instructions

Table C5-2 lists the Address translation instructions and their encodings. The syntax of the instructions includes `Xt`, that provides the address to be translated.

Table C5-2 Address translation instructions

op0	op1	CRn	CRm	op2	Mnemonic
0b01	0b000	0b0111	0b1000	0b000	AT S1E1R
0b01	0b000	0b0111	0b1000	0b001	AT S1E1W
0b01	0b000	0b0111	0b1000	0b010	AT S1E0R
0b01	0b000	0b0111	0b1000	0b011	AT S1E0W
0b01	0b000	0b0111	0b1001	0b000	AT S1E1RP
0b01	0b000	0b0111	0b1001	0b001	AT S1E1WP
0b01	0b100	0b0111	0b1000	0b000	AT S1E2R
0b01	0b100	0b0111	0b1000	0b001	AT S1E2W
0b01	0b100	0b0111	0b1000	0b100	AT S12E1R
0b01	0b100	0b0111	0b1000	0b101	AT S12E1W
0b01	0b100	0b0111	0b1000	0b110	AT S12E0R
0b01	0b100	0b0111	0b1000	0b111	AT S12E0W
0b01	0b110	0b0111	0b1000	0b000	AT S1E3R
0b01	0b110	0b0111	0b1000	0b001	AT S1E3W

For more information about these instructions, see Address translation instructions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.1.4.3 TLB maintenance instructions

Table C5-3 lists the TLB maintenance instructions and their encodings. Instructions that take an argument include `Xt` in the instruction syntax. For instructions that do not take an argument, the Xt field is encoded as `0b11111`. For these instructions, if the Xt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Xt field is set to `0b11111`.

Table C5-3 TLB maintenance instructions

op0	op1	CRn	CRm	op2	Mnemonic
0b01	0b000	0b1000	0b0001	0b000	TLBI VMALLE1OS
0b01	0b000	0b1000	0b0001	0b001	TLBI VAE1OS
0b01	0b000	0b1000	0b0001	0b010	TLBI ASIDE1OS
0b01	0b000	0b1000	0b0001	0b011	TLBI VAAE1OS
0b01	0b000	0b1000	0b0001	0b101	TLBI VALE1OS
0b01	0b000	0b1000	0b0001	0b111	TLBI VAALE1OS
0b01	0b000	0b1000	0b0010	0b001	TLBI RVAE1IS
0b01	0b000	0b1000	0b0010	0b011	TLBI RVAAE1IS
0b01	0b000	0b1000	0b0010	0b101	TLBI RVALE1IS
0b01	0b000	0b1000	0b0010	0b111	TLBI RVAALE1IS
0b01	0b000	0b1000	0b0011	0b000	TLBI VMALLE1IS
0b01	0b000	0b1000	0b0011	0b001	TLBI VAE1IS
0b01	0b000	0b1000	0b0011	0b010	TLBI ASIDE1IS
0b01	0b000	0b1000	0b0011	0b011	TLBI VAAE1IS
0b01	0b000	0b1000	0b0011	0b101	TLBI VALE1IS
0b01	0b000	0b1000	0b0011	0b111	TLBI VAALE1IS
0b01	0b000	0b1000	0b0101	0b001	TLBI RVAE1OS
0b01	0b000	0b1000	0b0101	0b011	TLBI RVAAE1OS
0b01	0b000	0b1000	0b0101	0b101	TLBI RVALE1OS
0b01	0b000	0b1000	0b0101	0b111	TLBI RVAALE1OS
0b01	0b000	0b1000	0b0110	0b001	TLBI RVAE1
0b01	0b000	0b1000	0b0110	0b011	TLBI RVAAE1
0b01	0b000	0b1000	0b0110	0b101	TLBI RVALE1
0b01	0b000	0b1000	0b0110	0b111	TLBI RVAALE1
0b01	0b000	0b1000	0b0111	0b000	TLBI VMALLE1
0b01	0b000	0b1000	0b0111	0b001	TLBI VAE1
0b01	0b000	0b1000	0b0111	0b010	TLBI ASIDE1
0b01	0b000	0b1000	0b0111	0b011	TLBI VAAE1
0b01	0b000	0b1000	0b0111	0b101	TLBI VALE1
0b01	0b000	0b1000	0b0111	0b111	TLBI VAALE1
0b01	0b100	0b1000	0b0000	0b001	TLBI IPAS2E1IS

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2	Mnemonic
0b01	0b100	0b1000	0b0000	0b010	TLBI RIPAS2E1IS
0b01	0b100	0b1000	0b0000	0b101	TLBI IPAS2LE1IS
0b01	0b100	0b1000	0b0000	0b110	TLBI RIPAS2LE1IS
0b01	0b100	0b1000	0b0001	0b000	TLBI ALLE2OS
0b01	0b100	0b1000	0b0001	0b001	TLBI VAE2OS
0b01	0b100	0b1000	0b0001	0b100	TLBI ALLE1OS
0b01	0b100	0b1000	0b0001	0b101	TLBI VALE2OS
0b01	0b100	0b1000	0b0001	0b110	TLBI VMALLS12E1OS
0b01	0b100	0b1000	0b0010	0b001	TLBI RVAE2IS
0b01	0b100	0b1000	0b0010	0b101	TLBI RVALE2IS
0b01	0b100	0b1000	0b0011	0b000	TLBI ALLE2IS
0b01	0b100	0b1000	0b0011	0b001	TLBI VAE2IS
0b01	0b100	0b1000	0b0011	0b100	TLBI ALLE1IS
0b01	0b100	0b1000	0b0011	0b101	TLBI VALE2IS
0b01	0b100	0b1000	0b0011	0b110	TLBI VMALLS12E1IS
0b01	0b100	0b1000	0b0100	0b000	TLBI IPAS2E1OS
0b01	0b100	0b1000	0b0100	0b001	TLBI IPAS2E1
0b01	0b100	0b1000	0b0100	0b010	TLBI RIPAS2E1
0b01	0b100	0b1000	0b0100	0b011	TLBI RIPAS2E1OS
0b01	0b100	0b1000	0b0100	0b100	TLBI IPAS2LE1OS
0b01	0b100	0b1000	0b0100	0b101	TLBI IPAS2LE1
0b01	0b100	0b1000	0b0100	0b110	TLBI RIPAS2LE1
0b01	0b100	0b1000	0b0100	0b111	TLBI RIPAS2LE1OS
0b01	0b100	0b1000	0b0101	0b001	TLBI RVAE2OS
0b01	0b100	0b1000	0b0101	0b101	TLBI RVALE2OS
0b01	0b100	0b1000	0b0110	0b001	TLBI RVAE2
0b01	0b100	0b1000	0b0110	0b101	TLBI RVALE2
0b01	0b100	0b1000	0b0111	0b000	TLBI ALLE2
0b01	0b100	0b1000	0b0111	0b001	TLBI VAE2
0b01	0b100	0b1000	0b0111	0b100	TLBI ALLE1
0b01	0b100	0b1000	0b0111	0b101	TLBI VALE2
0b01	0b100	0b1000	0b0111	0b110	TLBI VMALLS12E1
0b01	0b110	0b1000	0b0001	0b000	TLBI ALLE3OS
0b01	0b110	0b1000	0b0001	0b001	TLBI VAE3OS
0b01	0b110	0b1000	0b0001	0b101	TLBI VALE3OS
0b01	0b110	0b1000	0b0010	0b001	TLBI RVAE3IS
0b01	0b110	0b1000	0b0010	0b101	TLBI RVALE3IS
0b01	0b110	0b1000	0b0011	0b000	TLBI ALLE3IS
0b01	0b110	0b1000	0b0011	0b001	TLBI VAE3IS
0b01	0b110	0b1000	0b0011	0b101	TLBI VALE3IS

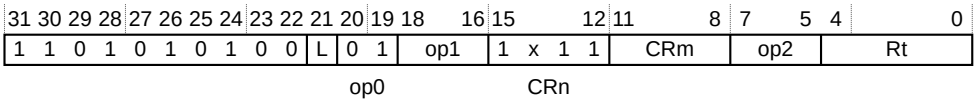
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2	Mnemonic
0b01	0b110	0b1000	0b0101	0b001	TLBI RVAE3OS
0b01	0b110	0b1000	0b0101	0b101	TLBI RVALE3OS
0b01	0b110	0b1000	0b0110	0b001	TLBI RVAE3
0b01	0b110	0b1000	0b0110	0b101	TLBI RVALE3
0b01	0b110	0b1000	0b0111	0b000	TLBI ALLE3
0b01	0b110	0b1000	0b0111	0b001	TLBI VAE3
0b01	0b110	0b1000	0b0111	0b101	TLBI VALE3

For more information about these instructions, see TLB maintenance instructions.

C5.1.4.4 Reserved encoding space for IMPLEMENTATION DEFINED instructions

The A64 instruction set reserves the following encoding space for IMPLEMENTATION DEFINED instructions:



The value of `L` defines the use of `Rt` as follows:

0 `Rt` is an argument supplied to the instruction.

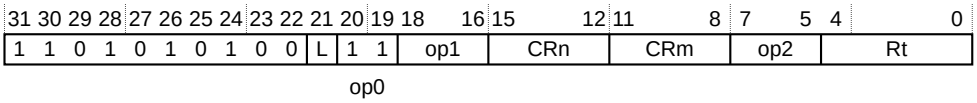
1 `Rt` is a result returned by the instruction.

IMPLEMENTATION DEFINED instructions in this encoding space are accessed using the `SYS` and `SYSL`.

See also Reserved encodings for IMPLEMENTATION DEFINED registers.

C5.1.5 `op0`== `0b11`, Moves to and from Special-purpose registers

The instructions that move data to and from non-debug System registers are encoded with `op0`== `0b11`, except that some of this encoding space is reserved for IMPLEMENTATION DEFINED functionality. The encoding of these instructions is:



C5.1.5.1 Instructions for accessing Special-purpose registers

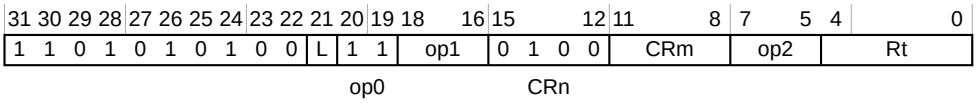
The value of `CRn` provides the next level of decode of these instructions. For Special-purpose registers, the value of CRn is 4.

The A64 instructions for accessing Special-purpose registers are:

```
MSR <Special-purpose_register>, Xt    ; Write to Special-purpose register
MRS Xt, <Special-purpose_register>    ; Read from Special-purpose register
```

Where `<Special-purpose_register>` is the register name, for example SP_EL0.

For these accesses, `CRn` has the value `0b0100`. The encoding for Special-purpose register accesses is:



Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The full list of Special-purpose registers is in Table C5-4. The characteristic of a Special-purpose register is that all direct and indirect reads and writes to the register appear to occur in program order relative to other instructions, without the need for any explicit synchronization.

Table C5-4 lists the encodings for `op1`, `CRm`, and `op2` fields for accesses to the Special-purpose registers in AArch64.

Table C5-4 Instruction encodings for Special-purpose register accesses

op1	CRn	CRm	op2	Access	Mnemonic	Accesses
0b000	0b0100	0b0000	0b000	RW	SPSR_EL1	SPSR_EL1
0b000	0b0100	0b0000	0b001	RW	ELR_EL1	ELR_EL1
0b000	0b0100	0b0001	0b000	RW	SP_EL0	SP_EL0
0b000	0b0100	0b0010	0b000	RW	SPSel	SPSel
0b000	0b0100	0b0010	0b010	RO	CurrentEL	CurrentEL
0b000	0b0100	0b0010	0b011	RW	PAN	PAN
0b000	0b0100	0b0010	0b100	RW	UAO	UAO
0b000	0b0100	0b0110	0b000	RW	ICC_PMR_EL1	ICC_PMR_EL1 ICV_PMR_EL1
0b011	0b0100	0b0010	0b000	RW	NZCV	NZCV
0b011	0b0100	0b0010	0b001	RW	DAIF	DAIF
0b011	0b0100	0b0010	0b101	RW	DIT	DIT
0b011	0b0100	0b0010	0b110	RW	SSBS	SSBS
0b011	0b0100	0b0100	0b000	RW	FPCR	FPCR
0b011	0b0100	0b0100	0b001	RW	FPSR	FPSR
0b011	0b0100	0b0101	0b000	RW	DSPSR_EL0	DSPSR_EL0
0b011	0b0100	0b0101	0b001	RW	DLR_EL0	DLR_EL0
0b100	0b0100	0b0000	0b000	RW	SPSR_EL2	SPSR_EL2
0b100	0b0100	0b0000	0b001	RW	ELR_EL2	ELR_EL2
0b100	0b0100	0b0001	0b000	RW	SP_EL1	SP_EL1
0b100	0b0100	0b0011	0b000	RW	SPSR_irq	SPSR_irq
0b100	0b0100	0b0011	0b001	RW	SPSR_abt	SPSR_abt
0b100	0b0100	0b0011	0b010	RW	SPSR_und	SPSR_und
0b100	0b0100	0b0011	0b011	RW	SPSR_fiq	SPSR_fiq
0b110	0b0100	0b0001	0b000	RW	SP_EL2	SP_EL2

All direct and indirect reads and writes to Special-purpose registers appear to occur in program order relative to other instructions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2 Special-purpose registers

This section describes the following Special-purpose registers:

- CurrentEL, that holds PSTATE.EL, and that software can read to determine the current Exception level.
- DAIF, that holds the current PSTATE.{D, A, I, F} interrupt mask bits.
- DIT, that holds the PSTATE.DIT bit.
- ELR_EL1, that holds the address to return to for an exception return from EL1.
- ELR_EL2, that holds the address to return to for an exception return from EL2.
- FPCR, that provides control of floating-point operation.
- FPSR, that provides floating-point status information.
- NZCV, that holds the PSTATE.{N, Z, C, V} condition flags.
- PAN, that holds the PSTATE.PAN state bit.
- SP_EL0, that holds the stack pointer for EL0.
- SP_EL1, that holds the stack pointer for EL1.
- SP_EL2, that holds the stack pointer for EL2.
- SPSel, that holds PSTATE.SP, that at EL1 or higher selects the current SP.
- SPSR_EL1, that holds process state on taking an exception to AArch64 EL1.
- SPSR_EL2, that holds process state on taking an exception to AArch64 EL2.
- SSBS, that holds the PSTATE.SSBS bit.
- UAO, that holds the PSTATE.UAO bit.

The following registers are also Special-purpose registers:

- DLR_EL0, that holds the address to return to for a return from Debug state.
- DSPSR_EL0, that holds process state on entry to Debug state.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.1 CurrentEL, Current Exception Level

The CurrentEL characteristics are:

Purpose

Holds the current Exception level.

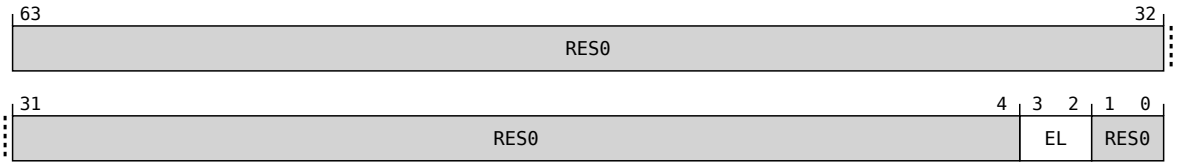
Configuration

There are no configuration notes.

Attributes

CurrentEL is a 64-bit register.

Field descriptions



Bits [63:4]

Reserved, RES0.

EL, bits [3:2]

Current Exception level.

EL	Meaning
0b00	EL0.
0b01	EL1.
0b10	EL2.
0b11	EL3.

When the Effective value of HCR_EL2.NV is 1, EL1 read accesses to the CurrentEL register return the value of 0b10 in this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to '10'.

Bits [1:0]

Reserved, RES0.

Accessing CurrentEL

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, CurrentEL

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0010	0b010

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    X[t, 64] = Zeros(60):PSTATE.EL:Zeros(2);
elsif PSTATE.EL == EL2 then
    X[t, 64] = Zeros(60):PSTATE.EL:Zeros(2);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.2 DAIF, Interrupt Mask Bits

The DAIF characteristics are:

Purpose

Allows access to the interrupt mask bits.

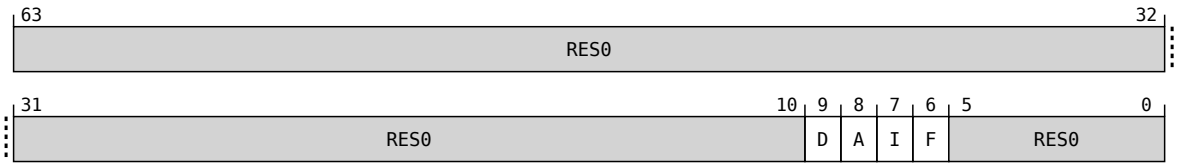
Configuration

There are no configuration notes.

Attributes

DAIF is a 64-bit register.

Field descriptions



Bits [63:10]

Reserved, RES0.

D, bit [9]

Process state D mask.

D	Meaning
0b0	Watchpoint, Breakpoint, and Software Step exceptions targeted at the current Exception level are not masked.
0b1	Watchpoint, Breakpoint, and Software Step exceptions targeted at the current Exception level are masked.

When the target Exception level of the debug exception is higher than the current Exception level, the exception is not masked by this bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to '1'.

A, bit [8]

SError exception mask bit.

A	Meaning
0b0	Exception not masked.
0b1	Exception masked.

The reset behavior of this field is:

- On a Warm reset, this field resets to '1'.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I, bit [7]

IRQ mask bit.

I	Meaning
0b0	Exception not masked.
0b1	Exception masked.

The reset behavior of this field is:

- On a Warm reset, this field resets to '1'.

F, bit [6]

FIQ mask bit.

F	Meaning
0b0	Exception not masked.
0b1	Exception masked.

The reset behavior of this field is:

- On a Warm reset, this field resets to '1'.

Bits [5:0]

Reserved, RES0.

Accessing DAIF

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, DAIF

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0010	0b001

```
if PSTATE.EL == EL0 then
    if SCTLR_EL1.UMA == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        X[t, 64] = Zeros(54):PSTATE.<D,A,I,F>:Zeros(6);
elsif PSTATE.EL == EL1 then
    X[t, 64] = Zeros(54):PSTATE.<D,A,I,F>:Zeros(6);
elsif PSTATE.EL == EL2 then
    X[t, 64] = Zeros(54):PSTATE.<D,A,I,F>:Zeros(6);
```

MSR DAIF, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0010	0b001

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if PSTATE.EL == EL0 then
    if SCTLR_EL1.UMA == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        PSTATE.<D,A,I,F> = X[t, 64]<9:6>;
elsif PSTATE.EL == EL1 then
    PSTATE.<D,A,I,F> = X[t, 64]<9:6>;
elsif PSTATE.EL == EL2 then
    PSTATE.<D,A,I,F> = X[t, 64]<9:6>;
```

MSR DAIFSet, #<imm>

op0	op1	CRn	op2
0b00	0b011	0b0100	0b110

MSR DAIFClr, #<imm>

op0	op1	CRn	op2
0b00	0b011	0b0100	0b111

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.3 DIT, Data Independent Timing

The DIT characteristics are:

Purpose

Allows access to the Data Independent Timing bit.

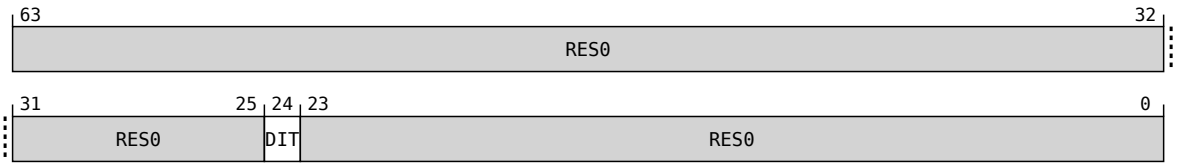
Configuration

This register is present only when FEAT_DIT is implemented. Otherwise, direct accesses to DIT are UNDEFINED.

Attributes

DIT is a 64-bit register.

Field descriptions



Bits [63:25]

Reserved, RES0.

DIT, bit [24]

Data Independent Timing.

DIT	Meaning
0b0	The architecture makes no statement about the timing properties of any instructions.
0b1	The architecture requires that the execution time of a data-independent-time sequence of code must be independent of all data-independent-time values. For more information, see About PSTATE.DIT.

The list of data-independent-time instructions is:

- Flag Manipulation: CFINV.
- Data Processing – Immediate:
  - Add/subtract (immediate): ADD, ADDS, SUB, and SUBS.
  - Bitfield: BFM, SBFM, and UBFM.
  - Extract: EXTR.
  - Logical (immediate): AND, ANDS, EOR, and ORR.
  - Min/max (immediate): SMAX, SMIN, UMAX, and UMIN.
- Data Processing – Register:
  - Add/subtract (extended register): ADD, ADDS, SUB, and SUBS.
  - Add/subtract (shifted register): ADD, ADDS, SUB, and SUBS.
  - Add/subtract (with carry): ADC, ADCS, SBC, and SBCS.
  - Conditional compare (immediate): CCMN, and CCMP.
  - Conditional compare (register): CCMN, and CCMP.
  - Conditional select: CSEL, CSINC, CSINV, and CSNEG.
  - Data-processing (1 source): ABS, CLS, CLZ, CNT, CTZ, RBIT, REV16, REV32, and REV.
  - Data-processing (2 source): ASRV, CRC32B, CRC32CB, CRC32CH, CRC32CW, CRC32CX, CRC32H, CRC32W, CRC32X, LSLV, LSRV, RORV, SMAX, SMIN, UMAX, and UMIN.
  - Data-processing (3 source): MADD, MSUB, SMADDL, SMSUBL, SMULH, UMADDL, UMSUBL, and UMULH.
  - Evaluate into flags: SETF16, and SETF8.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - Logical (shifted register): AND, ANDS, BIC, BICS, EON, EOR, ORN, and ORR.
  - Rotate right into flags: RMIF.
- Data Processing – Scalar Floating-Point and Advanced SIMD:
  - Advanced SIMD across lanes: ADDV, SADDLV, SMAXV, SMINV, UADDLV, UMAXV, and UMINV.
  - Advanced SIMD copy: DUP, INS, SMOV, and UMOV.
  - Advanced SIMD extract: EXT.
  - Advanced SIMD modified immediate: BIC, and ORR.
  - Advanced SIMD permute: TRN1, TRN2, UZP1, UZP2, ZIP1, and ZIP2.
  - Advanced SIMD scalar copy: DUP.
  - Advanced SIMD scalar pairwise: ADDP.
  - Advanced SIMD scalar shift by immediate: Advanced SIMD scalar shift by immediate: SHL, SLI, SRSHR, SRI, SRSRA, SSHR, SSRA, URSHR, URSRA, USHR, and USRA.
  - Advanced SIMD scalar three same: ADD, CMEQ, CMGE, CMGT, CMHI, CMHS, CMTST, SQDMULH, SQRDMULH, SSHL, SUB, and USHL.
  - Advanced SIMD scalar three same extra: SQRDMLAH.
  - Advanced SIMD scalar two-register miscellaneous: ABS, CMEQ, CMGE, CMGT, CMLE, CMLT, and NEG.
  - Advanced SIMD scalar x indexed element: SQDMULH, SQRDMLAH, and SQRDMULH.
  - Advanced SIMD shift by immediate: RSHRN2, RSHRN, SHL, SHRN2, SHRN, SLI, SRI, SSHLL2, SSHLL, SSHR, SSRA, USHLL2, USHLL, USHR, and USRA.
  - Advanced SIMD table lookup: LUTI2, LUTI4, TBL, and TBX.
  - Advanced SIMD three different: ADDHN2, ADDHN, PMULL2, PMULL, RADDHN2, RADDHN, RSUBHN2, RSUBHN, SABAL2, SABAL, SABDL2, SABDL, SADDL2, SADDL, SADDW2, SADDW, SMLAL2, SMLAL, SMLSL2, SMLSL, SMULL2, SMULL, SSUBL2, SSUBL, SSUBW2, SSUBW, SUBHN2, SUBHN, UABAL2, UABAL, UABDL2, UABDL, UADDL2, UADDL, UADDW2, UADDW, UMLAL2, UMLAL, UMLSL2, UMLSL, UMULL2, UMULL, USUBL2, USUBL, USUBW2, and USUBW.
  - Advanced SIMD three same: ADD, ADDP, AND, BIC, BIF, BIT, BSL, CMEQ, CMGE, CMGT, CMHI, CMHS, CMTST, EOR, MLA, MLS, MUL, ORN, ORR, PMUL, SABA, SABD, SHADD, SHSUB, SMAX, SMAXP, SMIN, SMINP, SQDMULH, SQRDMULH, SSHL, SUB, UABA, UABD, UHADD, UHSUB, UMAX, UMAXP, UMIN, UMINP, and USHL.
  - Advanced SIMD three-register extension: SDOT, SMMLA, SQRDMLAH, UDOT, UMMLA, USDOT, and USMMLA.
  - Advanced SIMD two-register miscellaneous: ABS, CLS, CLZ, CMEQ, CMGE, CMGT, CMLE, CMLT, CNT, NEG, NOT, RBIT, REV16, REV32, REV64, SADALP, SADDLP, SHLL2, SHLL, UADALP, UADDLP, XTN2, and XTN.
  - Advanced SIMD vector x indexed element: MLA, MLS, MUL, SDOT, SMLAL2, SMLAL, SMLSL2, SMLSL, SMULL2, SMULL, SQDMULH, SQRDMLAH, SQRDMULH, SUDOT, UDOT, UMLAL2, UMLAL, UMLSL2, UMLSL, UMULL2, UMULL, and USDOT.
  - Cryptographic AES: AESD, AESE, AESIMC, and AESMC.
  - Cryptographic four-register: BCAX, EOR3, and SM3SS1.
  - Cryptographic three-register SHA: SHA1C, SHA1M, SHA1P, SHA1SU0, SHA256H2, SHA256H, and SHA256SU1.
  - Cryptographic three-register SHA 512: RAX1, SHA512H2, SHA512H, SHA512SU1, SM3PARTW1, SM3PARTW2, and SM4EKEY.
  - Cryptographic three-register, imm2: SM3TT1A, SM3TT1B, SM3TT2A, and SM3TT2B.
  - Cryptographic three-register, imm6: XAR.
  - Cryptographic two-register SHA: SHA1H, SHA1SU1, and SHA256SU0.
  - Cryptographic two-register SHA 512: SHA512SU0, and SM4E.
  - Floating-point conditional select: FCSEL.
- Loads and Stores:
  - 128-bit atomic memory operations: LDCLRP, LDCLRPA, LDCLRPAL, LDCLRPL, LDSETP, LDSETPA, LDSETPAL, LDSETPL, SWPP, SWPPA, SWPPAL, and SWPPL.
  - Advanced SIMD load/store multiple structures: LD1, LD2, LD3, LD4, ST1, ST2, ST3, and ST4.
  - Advanced SIMD load/store multiple structures (post-indexed): LD1, LD2, LD3, LD4, ST1, ST2, ST3, and ST4.
  - Advanced SIMD load/store single structure: LD1, LD1R, LD2, LD2R, LD3, LD3R, LD4, LD4R, LDAP1, ST1, ST2, ST3, ST4, and STL1.
  - Advanced SIMD load/store single structure (post-indexed): LD1, LD1R, LD2, LD2R, LD3, LD3R, LD4, LD4R, ST1, ST2, ST3, and ST4.
  - Atomic memory operations: LDADD, LDADDA, LDADDAB, LDADDAH, LDADDAL, LDADDALB, LDADDALH, LDADDB, LDADDH, LDADDL, LDADDLB, LDADDLH, LDAPR, LDAPRB, LDAPRH, LDCLR, LDCLRA, LDCLRAB, LDCLRAH, LDCLRAL, LDCLRALB, LDCLRALH, LDCLRB, LDCLRH, LDCLRL, LDCLRLB, LDCLRLH, LDEOR, LDEORA, LDEORAB, LDEORAH, LDEORAL, LDEORALB, LDEORALH, LDEORB, LDEORH, LDEORL, LDEORLB, LDEORLH, LDSET, LDSETA, LDSETAB, LDSETAH, LDSETAL, LDSETALB, LDSETALH, LDSETB, LDSETH, LDSETL, LDSETLB, LDSETLH, LDSMAX, LDSMAXA, LDSMAXAB, LDSMAXAH, LDSMAXAL, LDSMAXALB, LDSMAXALH, LDSMAXB, LDSMAXH, LDSMAXL, LDSMAXLB, LDSMAXLH, LDSMIN, LDSMINA, LDSMINAB, LDSMINAH, LDSMINAL, LDSMINALB, LDSMINALH, LDSMINB, LDSMINH, LDSMINL, LDSMINLB, LDSMINLH, LDUMAX, LDUMAXA, LDUMAXAB, LDUMAXAH, LDUMAXAL, LDUMAXALB, LDUMAXALH, LDUMAXB, LDUMAXH, LDUMAXL, LDUMAXLB, LDUMAXLH, LDUMIN, LDUMINA, LDUMINAB, LDUMINAH, LDUMINAL, LDUMINALB, LDUMINALH, LDUMINB, LDUMINH, LDUMINL, LDUMINLB, and LDUMINLH.
  - LDAPR/STLR (SIMD&FP): LDAPUR, and STLUR.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- LDAPR/STLR (unscaled immediate): LDAPUR, LDAPURB, LDAPURH, LDAPURSB, LDAPURSH, LDAPURSW, STLUR, STLURB, and STLURH.
- LDAPR/STLR (writeback): LDAPR, and STLR.
- LDIAPP/STILP: LDIAPP, and STILP.
- Load/store exclusive pair: LDAXP, LDXP, STLXP, and STXP.
- Load/store exclusive register: LDAXR, LDAXRB, LDAXRH, LDXR, LDXRB, LDXRH, STLXR, STLXRB, STLXRH, STXR, STXRB and STXRH.
- Load/store no-allocate pair (offset): LDNP, and STNP.
- Load/store ordered: LDAR, LDARB, LDARH, LDLAR, LDLARB, LDLARH, STLLR, STLLRB, STLLRH, STLR, STLRB, and STLRH.
- Load/store register (immediate post-indexed): LDR, LDRB, LDRH, LDRSB, LDRSH, LDRSW, STR, STRB, and STRH.
- Load/store register (immediate pre-indexed): LDR, LDRB, LDRH, LDRSB, LDRSH, LDRSW, STR, STRB, and STRH.
- Load/store register (pac): LDRAA, and LDRAB.
- Load/store register (register offset): LDR, LDRB, LDRH, LDRSB, LDRSH, LDRSW, STR, STRB, and STRH.
- Load/store register (unprivileged): LDTR, LDTRB, LDTRH, LDTRSB, LDTRSH, LDTRSW, STTR, STTRB, and STTRH.
- Load/store register (unscaled immediate): LDUR, LDURB, LDURH, LDURSB, LDURSH, LDURSW, STUR, STURB, and STURH.
- Load/store register (unsigned immediate): LDR, LDRB, LDRH, LDRSB, LDRSH, LDRSW, STR, STRB, and STRH.
- Load/store register pair (offset): LDP, LDPSW, and STP.
- Load/store register pair (post-indexed): LDP, LDPSW, and STP.
- Load/store register pair (pre-indexed): LDP, LDPSW, and STP.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Bits [23:0]

Reserved, RES0.

Accessing DIT

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, DIT

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0010	0b101

```
if !IsFeatureImplemented(FEAT_DIT) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    X[t, 64] = Zeros(39):PSTATE.DIT:Zeros(24);
elsif PSTATE.EL == EL1 then
    X[t, 64] = Zeros(39):PSTATE.DIT:Zeros(24);
elsif PSTATE.EL == EL2 then
    X[t, 64] = Zeros(39):PSTATE.DIT:Zeros(24);
```

MSR DIT, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0010	0b101

```
if !IsFeatureImplemented(FEAT_DIT) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    PSTATE.DIT = X[t, 64]<24>;
elsif PSTATE.EL == EL1 then
    PSTATE.DIT = X[t, 64]<24>;
elsif PSTATE.EL == EL2 then
    PSTATE.DIT = X[t, 64]<24>;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR DIT, #<imm>
```

op0	op1	CRn	op2
0b00	0b011	0b0100	0b010

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.4 ELR_EL1, Exception Link Register (EL1)

The ELR_EL1 characteristics are:

Purpose

When taking an exception to EL1, holds the address to return to.

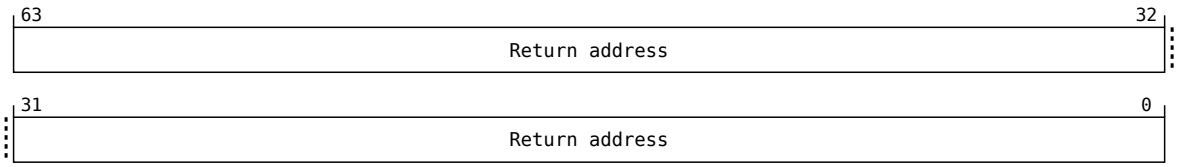
Configuration

There are no configuration notes.

Attributes

ELR_EL1 is a 64-bit register.

Field descriptions



ADDR, bits [63:0]

Return address.

An exception return from EL1 using AArch64 makes ELR_EL1 become UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing ELR_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, ELR_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    X[t, 64] = ELR_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ELR_EL1;
```

`MSR ELR_EL1, <Xt>`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0000	0b001

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    ELR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    ELR_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.5 ELR_EL2, Exception Link Register (EL2)

The ELR_EL2 characteristics are:

Purpose

When taking an exception to EL2, holds the address to return to.

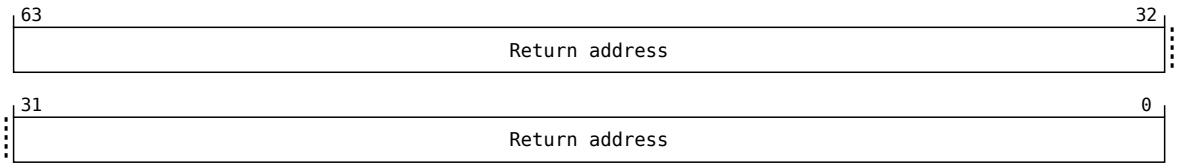
Configuration

There are no configuration notes.

Attributes

ELR_EL2 is a 64-bit register.

Field descriptions



ADDR, bits [63:0]

Return address.

An exception return from EL2 using AArch64 makes ELR_EL2 become UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing ELR_EL2

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, ELR_EL2`

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ELR_EL2;
```

`MSR ELR_EL2, <Xt>`

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0000	0b001

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    ELR_EL2 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.6 FPCR, Floating-point Control Register

The FPCR characteristics are:

Purpose

Controls behavior of the floating-point instructions.

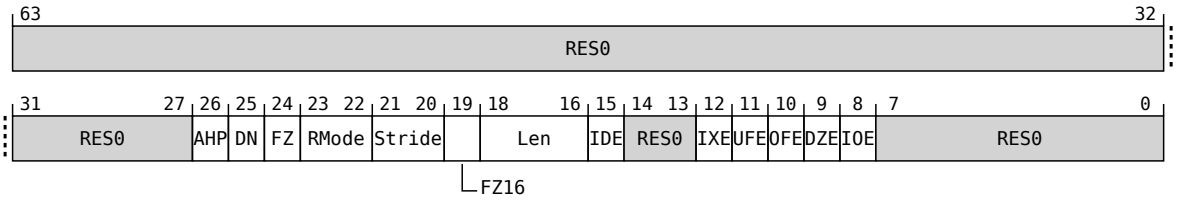
Configuration

There are no configuration notes.

Attributes

FPCR is a 64-bit register.

Field descriptions



Bits [63:27]

Reserved, RES0.

AHP, bit [26]

Alternative half-precision control bit.

AHP	Meaning
0b0	IEEE half-precision format selected.
0b1	Alternative half-precision format selected.

This bit is used only for conversions between half-precision floating-point and other floating-point formats.

The data-processing instructions added as part of the FEAT_FP16 extension always use the IEEE half-precision format, and ignore the value of this bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

DN, bit [25]

Default NaN use for NaN propagation.

DN	Meaning
0b0	NaN operands propagate through to the output of a floating-point operation.
0b1	Any operation involving one or more NaNs returns the Default NaN. This bit has no effect on the output of the FABS and FNEG instructions. This bit has no effect on the output of the FMAX, FMAXP, FMAXV, FMIN, FMINP, and FMINV instructions when FPCR.AH is 1.

The reset behavior of this field is:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

FZ, bit [24]

Flushing denormalized numbers to zero control bit.

FZ	Meaning
0b0	Flushing denormalized numbers to zero disabled.
0b1	Flushing denormalized numbers to zero enabled.

For more information, see 'Flushing denormalized numbers to zero' and the pseudocode of the floating-point instructions.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

RMode, bits [23:22]

Rounding Mode control field.

RMode	Meaning
0b00	Round to Nearest (RN) mode.
0b01	Round towards Plus Infinity (RP) mode.
0b10	Round towards Minus Infinity (RM) mode.
0b11	Round towards Zero (RZ) mode.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Stride, bits [21:20]

This field has no function in AArch64 state, and nonzero values are ignored during execution in AArch64 state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation implements FPSCR.LEN,STRIDE as RAZ, access to this field is RAZ/WI.

FZ16, bit [19]

When FEAT_FP16 is implemented:

Flushing denormalized numbers to zero control bit on half-precision data-processing instructions.

FZ16	Meaning
0b0	For some instructions, this bit disables flushing to zero of inputs and outputs that are half-precision denormalized numbers.
0b1	Flushing denormalized numbers to zero enabled.

For more information, see 'Flushing denormalized numbers to zero' and the pseudocode of the floating-point instructions.

The reset behavior of this field is:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Len, bits [18:16]

This field has no function in AArch64 state, and nonzero values are ignored during execution in AArch64 state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation implements FPSCR.LEN,STRIDE as RAZ, access to this field is RAZ/WI.

IDE, bit [15]

Input Denormal floating-point exception trap enable.

IDE	Meaning
0b0	Untrapped exception handling selected. If the floating-point exception occurs, the FPSR.IDC bit is set to 1.
0b1	Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the FPSR.IDC bit.

The value of this bit controls both scalar and vector floating-point arithmetic.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement trapping of Input Denormal floating-point exceptions, access to this field is RAZ/WI.

Bits [14:13]

Reserved, RES0.

IXE, bit [12]

Inexact floating-point exception trap enable.

IXE	Meaning
0b0	Untrapped exception handling selected. If the floating-point exception occurs, the FPSR.IXC bit is set to 1.
0b1	Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the FPSR.IXC bit.

The value of this bit controls both scalar and vector floating-point arithmetic.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement trapping of Inexact floating-point exceptions, access to this field is RAZ/WI.

UFE, bit [11]

Underflow floating-point exception trap enable.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

UFE	Meaning
0b0	Untrapped exception handling selected. If the floating-point exception occurs, the FPSR.UFC bit is set to 1.
0b1	Trapped exception handling selected. If the floating-point exception occurs and Flush-to-zero is not enabled, the PE does not update the FPSR.UFC bit.

The value of this bit controls both scalar and vector floating-point arithmetic.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement trapping of Underflow floating-point exceptions, access to this field is RAZ/WI.

OFE, bit [10]

Overflow floating-point exception trap enable.

OFE	Meaning
0b0	Untrapped exception handling selected. If the floating-point exception occurs, the FPSR.OFC bit is set to 1.
0b1	Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the FPSR.OFC bit.

The value of this bit controls both scalar and vector floating-point arithmetic.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement trapping of Overflow floating-point exceptions, access to this field is RAZ/WI.

DZE, bit [9]

Divide by Zero floating-point exception trap enable.

DZE	Meaning
0b0	Untrapped exception handling selected. If the floating-point exception occurs, the FPSR.DZC bit is set to 1.
0b1	Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the FPSR.DZC bit.

The value of this bit controls both scalar and vector floating-point arithmetic.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement trapping of Divide by Zero floating-point exceptions, access to this field is RAZ/WI.

IOE, bit [8]

Invalid Operation floating-point exception trap enable.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IOE	Meaning
0b0	Untrapped exception handling selected. If the floating-point exception occurs, the FPSR.IOC bit is set to 1.
0b1	Trapped exception handling selected. If the floating-point exception occurs, the PE does not update the FPSR.IOC bit.

The value of this bit controls both scalar and vector floating-point arithmetic.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When an implementation does not implement trapping of Invalid Operation floating-point exceptions, access to this field is RAZ/WI.

Bits [7:0]

Reserved, RES0.

Accessing FPCR

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, FPCR`

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0100	0b000

```
if PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CPACR_EL1.FPEN != '11' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x00);
        else
            AArch64.SystemAccessTrap(EL1, 0x07);
    elsif !ELIsInHost(EL2) && CPTR_EL2.TFP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    else
        X[t, 64] = FPCR;
elsif PSTATE.EL == EL1 then
    if CPACR_EL1.FPEN IN {'x0'} then
        AArch64.SystemAccessTrap(EL1, 0x07);
    elsif !ELIsInHost(EL2) && CPTR_EL2.TFP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif ELIsInHost(EL2) then
        AArch64.SystemAccessTrap(EL2, 0x07);
    else
        X[t, 64] = FPCR;
elsif PSTATE.EL == EL2 then
    if !ELIsInHost(EL2) && CPTR_EL2.TFP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    else
        X[t, 64] = FPCR;
```

`MSR FPCR, <Xt>`

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0100	0b000

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CPACR_EL1.FPEN != '11' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x00);
        else
            AArch64.SystemAccessTrap(EL1, 0x07);
    elsif !ELIsInHost(EL2) && CPTR_EL2.TFP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    else
        FPCR = X[t, 64];
elsif PSTATE.EL == EL1 then
    if CPACR_EL1.FPEN IN {'x0'} then
        AArch64.SystemAccessTrap(EL1, 0x07);
    elsif !ELIsInHost(EL2) && CPTR_EL2.TFP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif ELIsInHost(EL2) then
        AArch64.SystemAccessTrap(EL2, 0x07);
    else
        FPCR = X[t, 64];
elsif PSTATE.EL == EL2 then
    if !ELIsInHost(EL2) && CPTR_EL2.TFP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    else
        FPCR = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.7 FPSR, Floating-point Status Register

The FPSR characteristics are:

Purpose

Provides floating-point system status information.

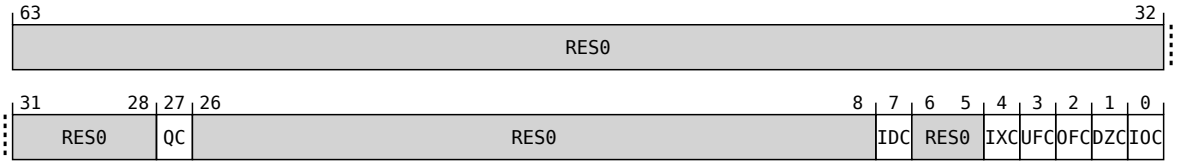
Configuration

There are no configuration notes.

Attributes

FPSR is a 64-bit register.

Field descriptions



Bits [63:28]

Reserved, RES0.

QC, bit [27]

Cumulative saturation bit, Advanced SIMD only. This bit is set to 1 to indicate that an Advanced SIMD integer operation has saturated since 0 was last written to this bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [26:8]

Reserved, RES0.

IDC, bit [7]

Input Denormal cumulative floating-point exception bit. This bit is set to 1 to indicate that the Input Denormal floating-point exception has occurred since 0 was last written to this bit.

How scalar and Advanced SIMD floating-point instructions update this bit depends on the value of the FPCR.IDE bit. This bit is set to 1 to indicate a floating-point exception only if FPCR.IDE is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [6:5]

Reserved, RES0.

IXC, bit [4]

Inexact cumulative floating-point exception bit. This bit is set to 1 to indicate that the Inexact floating-point exception has occurred since 0 was last written to this bit.

How scalar and Advanced SIMD floating-point instructions update this bit depends on the value of the FPCR.IXE bit. This bit is set to 1 to indicate a floating-point exception only if FPCR.IXE is 0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The criteria for the Inexact floating-point exception to occur are affected by whether denormalized numbers are flushed to zero and by the value of the FPCR.AH bit. For more information, see 'Floating-point exceptions and exception traps'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

UFC, bit [3]

Underflow cumulative floating-point exception bit. This bit is set to 1 to indicate that the Underflow floating-point exception has occurred since 0 was last written to this bit.

How scalar and Advanced SIMD floating-point instructions update this bit depends on the value of the FPCR.UFE bit. This bit is set to 1 to indicate a floating-point exception only if FPCR.UFE is 0 or if flushing denormalized numbers to zero is enabled.

The criteria for the Underflow floating-point exception to occur are affected by whether denormalized numbers are flushed to zero and by the value of the FPCR.AH bit. For more information, see 'Floating-point exceptions and exception traps'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

OFC, bit [2]

Overflow cumulative floating-point exception bit. This bit is set to 1 to indicate that the Overflow floating-point exception has occurred since 0 was last written to this bit.

How scalar and Advanced SIMD floating-point instructions update this bit depends on the value of the FPCR.OFE bit. This bit is set to 1 to indicate a floating-point exception only if FPCR.OFE is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

DZC, bit [1]

Divide by Zero cumulative floating-point exception bit. This bit is set to 1 to indicate that the Divide by Zero floating-point exception has occurred since 0 was last written to this bit.

How scalar and Advanced SIMD floating-point instructions update this bit depends on the value of the FPCR.DZE bit. This bit is set to 1 to indicate a floating-point exception only if FPCR.DZE is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IOC, bit [0]

Invalid Operation cumulative floating-point exception bit. This bit is set to 1 to indicate that the Invalid Operation floating-point exception has occurred since 0 was last written to this bit.

How scalar and Advanced SIMD floating-point instructions update this bit depends on the value of the FPCR.IOE bit. This bit is set to 1 to indicate a floating-point exception only if FPCR.IOE is 0.

The criteria for the Invalid Operation floating-point exception to occur are affected by the value of the FPCR.AH bit. For more information, see 'Floating-point exceptions and exception traps'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing FPSR

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, FPSR
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0100	0b001

```
if PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CPACR_EL1.FPEN != '11' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x00);
        else
            AArch64.SystemAccessTrap(EL1, 0x07);
    elsif !ELIsInHost(EL2) && CPTR_EL2.TFP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    else
        X[t, 64] = FPSR;
elsif PSTATE.EL == EL1 then
    if CPACR_EL1.FPEN IN {'x0'} then
        AArch64.SystemAccessTrap(EL1, 0x07);
    elsif !ELIsInHost(EL2) && CPTR_EL2.TFP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif ELIsInHost(EL2) then
        AArch64.SystemAccessTrap(EL2, 0x07);
    else
        X[t, 64] = FPSR;
elsif PSTATE.EL == EL2 then
    if !ELIsInHost(EL2) && CPTR_EL2.TFP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    else
        X[t, 64] = FPSR;
```

MSR FPSR, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0100	0b001

```
if PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CPACR_EL1.FPEN != '11' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x00);
        else
            AArch64.SystemAccessTrap(EL1, 0x07);
    elsif !ELIsInHost(EL2) && CPTR_EL2.TFP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    else
        FPSR = X[t, 64];
elsif PSTATE.EL == EL1 then
    if CPACR_EL1.FPEN IN {'x0'} then
        AArch64.SystemAccessTrap(EL1, 0x07);
    elsif !ELIsInHost(EL2) && CPTR_EL2.TFP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    elsif ELIsInHost(EL2) then
        AArch64.SystemAccessTrap(EL2, 0x07);
    else
        FPSR = X[t, 64];
elsif PSTATE.EL == EL2 then
    if !ELIsInHost(EL2) && CPTR_EL2.TFP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x07);
    else
        FPSR = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.8 NZCV, Condition Flags

The NZCV characteristics are:

Purpose

Allows access to the condition flags.

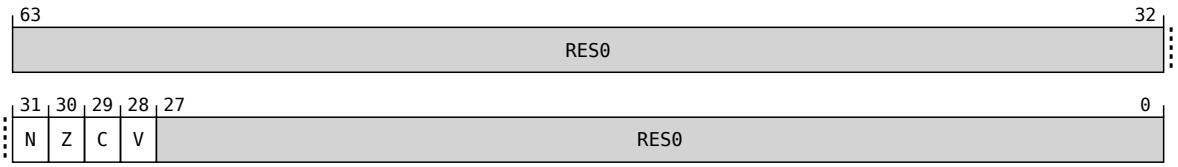
Configuration

There are no configuration notes.

Attributes

NZCV is a 64-bit register.

Field descriptions



Bits [63:32]

Reserved, RES0.

N, bit [31]

Negative condition flag. Set to 1 if the result of the last flag-setting instruction was negative.

Z, bit [30]

Zero condition flag. Set to 1 if the result of the last flag-setting instruction was zero, and to 0 otherwise. A result of zero often indicates an equal result from a comparison.

C, bit [29]

Carry condition flag. Set to 1 if the last flag-setting instruction resulted in a carry condition, for example an unsigned overflow on an addition.

V, bit [28]

Overflow condition flag. Set to 1 if the last flag-setting instruction resulted in an overflow condition, for example a signed overflow on an addition.

Bits [27:0]

Reserved, RES0.

Accessing NZCV

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, NZCV
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0010	0b000

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if PSTATE.EL == EL0 then
    X[t, 64] = Zeros(32):PSTATE.<N,Z,C,V>:Zeros(28);
elsif PSTATE.EL == EL1 then
    X[t, 64] = Zeros(32):PSTATE.<N,Z,C,V>:Zeros(28);
elsif PSTATE.EL == EL2 then
    X[t, 64] = Zeros(32):PSTATE.<N,Z,C,V>:Zeros(28);
```

MSR NZCV, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0010	0b000

```
if PSTATE.EL == EL0 then
    PSTATE.<N,Z,C,V> = X[t, 64]<31:28>;
elsif PSTATE.EL == EL1 then
    PSTATE.<N,Z,C,V> = X[t, 64]<31:28>;
elsif PSTATE.EL == EL2 then
    PSTATE.<N,Z,C,V> = X[t, 64]<31:28>;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.9 PAN, Privileged Access Never

The PAN characteristics are:

Purpose

Allows access to the Privileged Access Never bit.

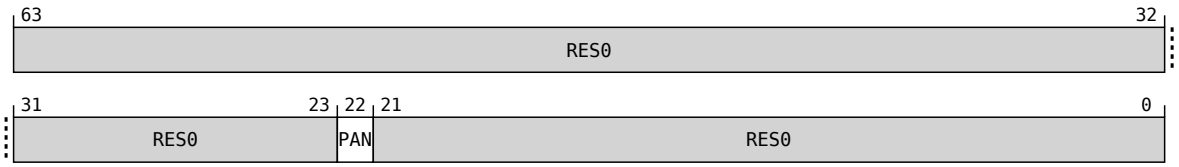
Configuration

This register is present only when FEAT_PAN is implemented. Otherwise, direct accesses to PAN are UNDEFINED.

Attributes

PAN is a 64-bit register.

Field descriptions



Bits [63:23]

Reserved, RES0.

PAN, bit [22]

Privileged Access Never.

PAN	Meaning
0b0	Privileged reads and write are not disabled by this mechanism.
0b1	Disables privileged read and write accesses to addresses accessible at EL0 for an enabled stage 1 translation regime that defines the EL0 permissions.

When the target of the exception is EL1 and the value of the SCTLR_EL1.SPAN bit is 0, this bit is set to 1. Otherwise, the value of this bit is preserved on taking an exception.

Bits [21:0]

Reserved, RES0.

Accessing PAN

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PAN
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0010	0b011

```
if !IsFeatureImplemented(FEAT_PAN) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL1 then
    X[t, 64] = Zeros(41):PSTATE.PAN:Zeros(22);
elsif PSTATE.EL == EL2 then
    X[t, 64] = Zeros(41):PSTATE.PAN:Zeros(22);
```

MSR PAN, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0010	0b011

```
if !IsFeatureImplemented(FEAT_PAN) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    PSTATE.PAN = X[t, 64]<22>;
elsif PSTATE.EL == EL2 then
    PSTATE.PAN = X[t, 64]<22>;
```

MSR PAN, #<imm>

op0	op1	CRn	op2
0b00	0b000	0b0100	0b100

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.10 SP_EL0, Stack Pointer (EL0)

The SP_EL0 characteristics are:

Purpose

Holds the stack pointer associated with EL0. At higher Exception levels, this is used as the current stack pointer when the value of SPSel.SP is 0.

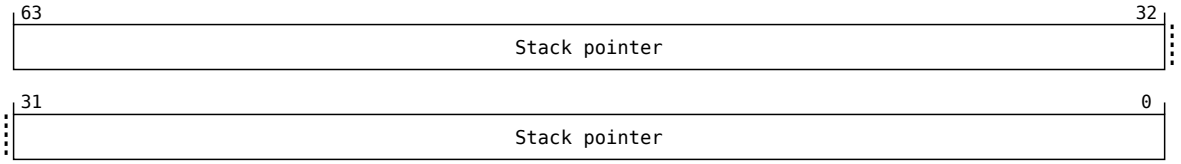
Configuration

There are no configuration notes.

Attributes

SP_EL0 is a 64-bit register.

Field descriptions



StackPointer, bits [63:0]

Stack pointer.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SP_EL0

When the value of PSTATE.SP is 0, this register is accessible at all Exception levels as the current stack pointer.

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, SP_EL0`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if PSTATE.SP == '0' then
        UNDEFINED;
    else
        X[t, 64] = SP_EL0;
elsif PSTATE.EL == EL2 then
    if PSTATE.SP == '0' then
        UNDEFINED;
    else
        X[t, 64] = SP_EL0;
```

`MSR SP_EL0, <Xt>`

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if PSTATE.SP == '0' then
        UNDEFINED;
    else
        SP_EL0 = X[t, 64];
elsif PSTATE.EL == EL2 then
    if PSTATE.SP == '0' then
        UNDEFINED;
    else
        SP_EL0 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.11 SP_EL1, Stack Pointer (EL1)

The SP_EL1 characteristics are:

Purpose

Holds the stack pointer associated with EL1. The value of SPSel.SP determines the current stack pointer.

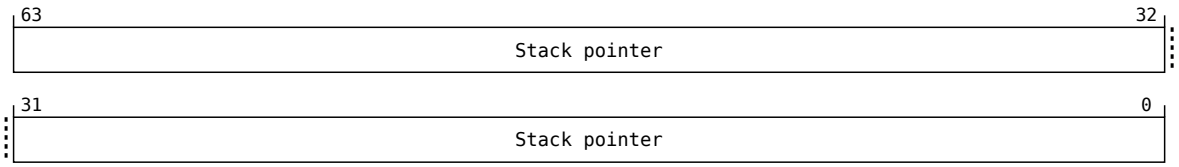
Configuration

There are no configuration notes.

Attributes

SP_EL1 is a 64-bit register.

Field descriptions



StackPointer, bits [63:0]

Stack pointer.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SP_EL1

This accessibility information only applies to accesses using the MRS or MSR instructions.

When the value of SPSel.SP is 1, this register is also accessible at EL1 as the current stack pointer.

Note

When the value of SPSel.SP is 0, SP_EL0 is used as the current stack pointer at all Exception levels.

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, SP_EL1`

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = SP_EL1;
```

`MSR SP_EL1, <Xt>`

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    SP_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.12 SP_EL2, Stack Pointer (EL2)

The SP_EL2 characteristics are:

Purpose

Holds the stack pointer associated with EL2. The value of SPSel.SP determines the current stack pointer.

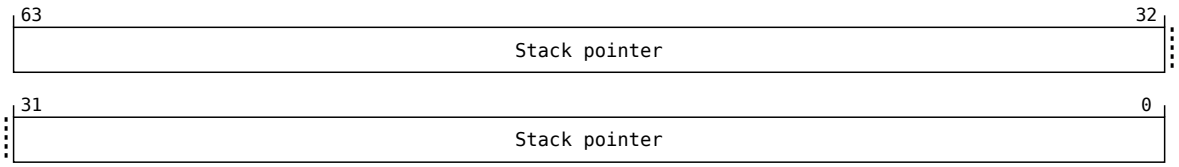
Configuration

There are no configuration notes.

Attributes

SP_EL2 is a 64-bit register.

Field descriptions



StackPointer, bits [63:0]

Stack pointer.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SP_EL2

This accessibility information only applies to accesses using the MRS or MSR instructions.

When the value of SPSel.SP is 1, this register is also accessible at EL2 as the current stack pointer.

Note

When the value of SPSel.SP is 0, SP_EL0 is used as the current stack pointer at all Exception levels.

This system instruction encoding is Undefined()

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SP_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b110	0b0100	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
```

```
MSR SP_EL2, <Xt>
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b110	0b0100	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.13 SPSel, Stack Pointer Select

The SPSel characteristics are:

Purpose

Allows the Stack Pointer to be selected between SP_EL0 and SP_ELx.

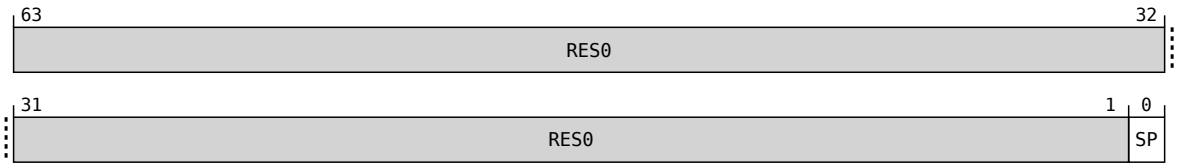
Configuration

There are no configuration notes.

Attributes

SPSel is a 64-bit register.

Field descriptions



Bits [63:1]

Reserved, RES0.

SP, bit [0]

Stack pointer to use. Possible values of this bit are:

SP	Meaning
0b0	Use SP_EL0 at all Exception levels.
0b1	Use SP_ELx for Exception level ELx.

The reset behavior of this field is:

- On a Warm reset, this field resets to '1'.

Accessing SPSel

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, SPSel

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    X[t, 64] = Zeros(63):PSTATE.SP;
elsif PSTATE.EL == EL2 then
    X[t, 64] = Zeros(63):PSTATE.SP;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MSR SPSel, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    PSTATE.SP = X[t, 64]<0>;
elsif PSTATE.EL == EL2 then
    PSTATE.SP = X[t, 64]<0>;
```

MSR SPSel, #<imm>

op0	op1	CRn	op2
0b00	0b000	0b0100	0b101

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.14 SPSR_EL1, Saved Program Status Register (EL1)

The SPSR_EL1 characteristics are:

Purpose

Holds the saved process state when an exception is taken to EL1.

Configuration

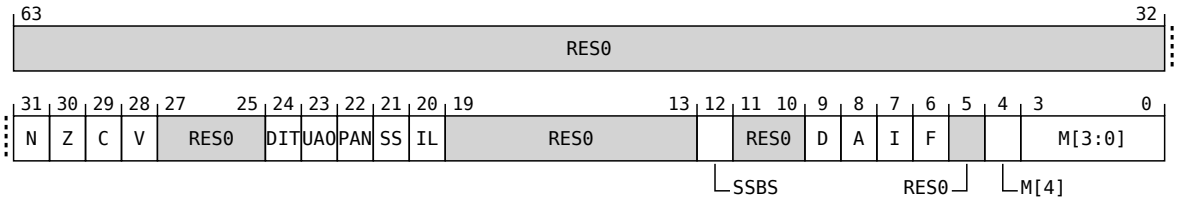
There are no configuration notes.

Attributes

SPSR_EL1 is a 64-bit register.

Field descriptions

When exception taken from AArch64 state:



An exception return from EL1 using AArch64 makes SPSR_EL1 become UNKNOWN.

Bits [63:32]

Reserved, RES0.

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to EL1, and copied to PSTATE.N on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to EL1, and copied to PSTATE.Z on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to EL1, and copied to PSTATE.C on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to EL1, and copied to PSTATE.V on executing an exception return operation in EL1.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [27:25]

Reserved, RES0.

DIT, bit [24]

When FEAT_DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to EL1, and copied to PSTATE.DIT on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

UAO, bit [23]

When FEAT_UAO is implemented:

User Access Override. Set to the value of PSTATE.UAO on taking an exception to EL1, and copied to PSTATE.UAO on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When FEAT_PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to EL1, and copied to PSTATE.PAN on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SS, bit [21]

Software Step. Set to the value of PSTATE.SS on taking an exception to EL1, and conditionally copied to PSTATE.SS on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to EL1, and copied to PSTATE.IL on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [19:13]

Reserved, RES0.

SSBS, bit [12]

When FEAT_SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to EL1, and copied to PSTATE.SSBS on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [11:10]

Reserved, RES0.

D, bit [9]

Debug exception mask. Set to the value of PSTATE.D on taking an exception to EL1, and copied to PSTATE.D on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]

SError exception mask. Set to the value of PSTATE.A on taking an exception to EL1, and copied to PSTATE.A on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to EL1, and copied to PSTATE.I on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to EL1, and copied to PSTATE.F on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [5]

Reserved, RES0.

M[4], bit [4]

Execution state. Set to `0b0`, the value of PSTATE.nRW, on taking an exception to EL1 from AArch64 state, and copied to PSTATE.nRW on executing an exception return operation in EL1.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

M[4]	Meaning
0b0	AArch64 execution state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[3:0], bits [3:0]

AArch64 Exception level and selected Stack Pointer.

M[3:0]	Meaning
0b0000	EL0.
0b0100	EL1 with SP_EL0 (EL1t).
0b0101	EL1 with SP_EL1 (EL1h).
0b1000	EL1 with SP_EL0 (EL1t) and either HCR_EL2.{NV, NV1} is {1,0} or HCR_EL2.{NV, NV2} is {1,1}.
0b1001	EL1 with SP_EL1 (EL1h) and either HCR_EL2.{NV, NV1} is {1,0} or HCR_EL2.{NV, NV2} is {1,1}.

Other values are reserved. If SPSR_EL1.M[3:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in EL1 is an illegal return event, as described in 'Illegal return events from AArch64 state'.

The bits in this field are interpreted as follows:

- M[3:2]: On an exception to EL1:
  - If the exception is taken from EL0:
    - M[3:2] is set to the value of PSTATE.EL on taking an exception to EL1.
  - If the exception is taken from EL1:
    - If the Effective value of HCR_EL2.{NV, NV1} is not {1,0} and the Effective value of HCR_EL2.{NV, NV2} is not {1,1}, then M[3:2] is set to the value of PSTATE.EL on taking an exception to EL1.
    - If the Effective value of HCR_EL2.{NV, NV1} is {1,0} or if the Effective value of HCR_EL2.{NV, NV2} is {1,1}, then M[3:2] is set to 0b10.
  - M[3:2] is copied to PSTATE.EL on executing a legal exception return operation in EL1.
- M[1] is unused and is 0 for all non-reserved values.
- M[0] is set to the value of PSTATE.SP on taking an exception to EL1 and copied to PSTATE.SP on executing an exception return operation in EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SPSR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, SPSR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    X[t, 64] = SPSR_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = SPSR_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR SPSR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    SPSR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    SPSR_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.15 SPSR_EL2, Saved Program Status Register (EL2)

The SPSR_EL2 characteristics are:

Purpose

Holds the saved process state when an exception is taken to EL2.

Configuration

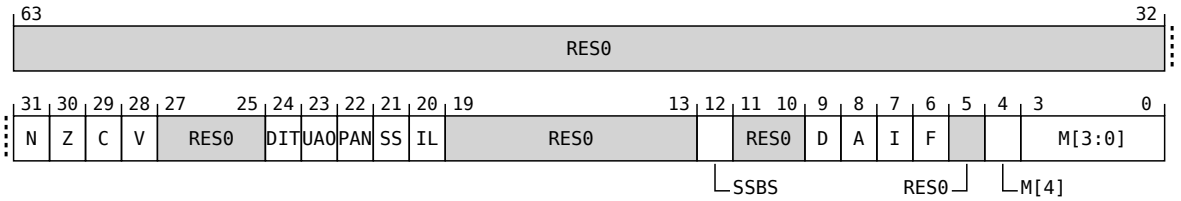
There are no configuration notes.

Attributes

SPSR_EL2 is a 64-bit register.

Field descriptions

When exception taken from AArch64 state:



An exception return from EL2 using AArch64 makes SPSR_EL2 become UNKNOWN.

Bits [63:32]

Reserved, RES0.

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to EL2, and copied to PSTATE.N on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to EL2, and copied to PSTATE.Z on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to EL2, and copied to PSTATE.C on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to EL2, and copied to PSTATE.V on executing an exception return operation in EL2.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [27:25]

Reserved, RES0.

DIT, bit [24]

When FEAT_DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on taking an exception to EL2, and copied to PSTATE.DIT on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

UAO, bit [23]

When FEAT_UAO is implemented:

User Access Override. Set to the value of PSTATE.UAO on taking an exception to EL2, and copied to PSTATE.UAO on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When FEAT_PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to EL2, and copied to PSTATE.PAN on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SS, bit [21]

Software Step. Set to the value of PSTATE.SS on taking an exception to EL2, and conditionally copied to PSTATE.SS on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to EL2, and copied to PSTATE.IL on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [19:13]

Reserved, RES0.

SSBS, bit [12]

When FEAT_SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to EL2, and copied to PSTATE.SSBS on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [11:10]

Reserved, RES0.

D, bit [9]

Debug exception mask. Set to the value of PSTATE.D on taking an exception to EL2, and copied to PSTATE.D on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]

SError exception mask. Set to the value of PSTATE.A on taking an exception to EL2, and copied to PSTATE.A on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to EL2, and copied to PSTATE.I on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to EL2, and copied to PSTATE.F on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [5]

Reserved, RES0.

M[4], bit [4]

Execution state. Set to `0b0`, the value of PSTATE.nRW, on taking an exception to EL2 from AArch64 state, and copied to PSTATE.nRW on executing an exception return operation in EL2.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

M[4]	Meaning
0b0	AArch64 execution state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[3:0], bits [3:0]

AArch64 Exception level and selected Stack Pointer.

M[3:0]	Meaning
0b0000	EL0.
0b0100	EL1 with SP_EL0 (EL1t).
0b0101	EL1 with SP_EL1 (EL1h).
0b1000	EL2 with SP_EL0 (EL2t).
0b1001	EL2 with SP_EL2 (EL2h).

Other values are reserved. If SPSR_EL2.M[3:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in EL2 is an illegal return event, as described in 'Illegal return events from AArch64 state'.

The bits in this field are interpreted as follows:

- M[3:2] is set to the value of PSTATE.EL on taking an exception to EL2 and copied to PSTATE.EL on executing an exception return operation in EL2.
- M[1] is unused and is 0 for all non-reserved values.
- M[0] is set to the value of PSTATE.SP on taking an exception to EL2 and copied to PSTATE.SP on executing an exception return operation in EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SPSR_EL2

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, SPSR_EL2`

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = SPSR_EL2;
```

`MSR SPSR_EL2, <Xt>`

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0000	0b000

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    SPSR_EL2 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.2.16 SSBS, Speculative Store Bypass Safe

The SSBS characteristics are:

Purpose

Allows access to the Speculative Store Bypass Safe bit.

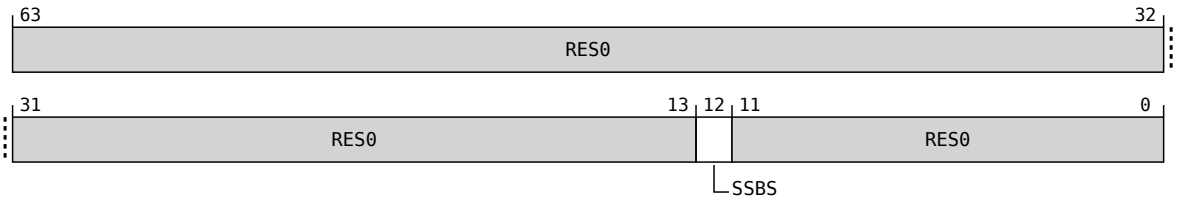
Configuration

This register is present only when FEAT_SSBS2 is implemented. Otherwise, direct accesses to SSBS are UNDEFINED.

Attributes

SSBS is a 64-bit register.

Field descriptions



Bits [63:13]

Reserved, RES0.

SSBS, bit [12]

Speculative Store Bypass Safe.

Prohibits speculative loads or stores which might practically allow a cache timing side channel.

A speculative value in a register is used in a potentially speculatively exploitable manner if it is used to form an address, generate condition codes, or generate SVE predicate values to be used by other instructions in the speculative sequence or if the execution timing of any other instructions in the speculative sequence is a function of the data loaded under speculation.

SSBS	Meaning
0b0	Hardware is not permitted to use speculative register values in a potentially speculatively exploitable manner if the speculative read that loads the register is from earlier in the coherence order than the entry generated by the latest store to that location using the same virtual address as the load instruction.
0b1	When the value of PSTATE.SSBS is 1, hardware is permitted to use speculative register values in a potentially speculatively exploitable manner if the speculative read that loads the register is from earlier in the coherence order than the entry generated by the latest store to that location using the same virtual address as the load instruction.

The value of this bit is set to the value in the SCTLR_ELx.DSSBS field on taking an exception to ELx.

The reset behavior of this field is:

- On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

Bits [11:0]

Reserved, RES0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing SSBS

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, SSBS`

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0010	0b110

```
if !IsFeatureImplemented(FEAT_SSBS2) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    X[t, 64] = Zeros(51):PSTATE.SSBS:Zeros(12);
elsif PSTATE.EL == EL1 then
    X[t, 64] = Zeros(51):PSTATE.SSBS:Zeros(12);
elsif PSTATE.EL == EL2 then
    X[t, 64] = Zeros(51):PSTATE.SSBS:Zeros(12);
```

`MSR SSBS, <Xt>`

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0010	0b110

```
if !IsFeatureImplemented(FEAT_SSBS2) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    PSTATE.SSBS = X[t, 64]<12>;
elsif PSTATE.EL == EL1 then
    PSTATE.SSBS = X[t, 64]<12>;
elsif PSTATE.EL == EL2 then
    PSTATE.SSBS = X[t, 64]<12>;
```

`MSR SSBS, #<imm>`

op0	op1	CRn	op2
0b00	0b011	0b0100	0b001

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3 A64 System instructions for cache maintenance

This section lists the A64 System instructions for cache maintenance.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.1 DC CISW, Data or unified Cache line Clean and Invalidate by Set/Way

The DC CISW characteristics are:

Purpose

Clean and Invalidate data cache by set/way.

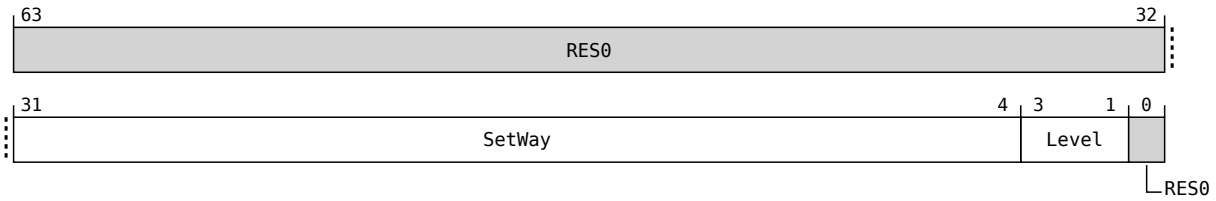
Configuration

There are no configuration notes.

Attributes

DC CISW is a 64-bit System instruction.

Field descriptions



Bits [63:32]

Reserved, RES0.

SetWay, bits [31:4]

Contains two fields:

- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

$A = Log_2(ASSOCIATIVITY)$, $L = Log_2(LINELEN)$, $B = (L + S)$, $S = Log_2(NSETS)$.

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

Level, bits [3:1]

Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

Bit [0]

Reserved, RES0.

Executing DC CISW

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED.
- The instruction performs cache maintenance on one of:
  - No cache lines.
  - A single arbitrary cache line.
  - Multiple arbitrary cache lines.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`DC CISW, <Xt>`

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b1110	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TSW == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.DC(X[t, 64], CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_SetWay);
elsif PSTATE.EL == EL2 then
    AArch64.DC(X[t, 64], CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_SetWay);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.2 DC CIVAC, Data or unified Cache line Clean and Invalidate by VA to PoC

The DC CIVAC characteristics are:

Purpose

Clean and Invalidate data cache by address to Point of Coherency.

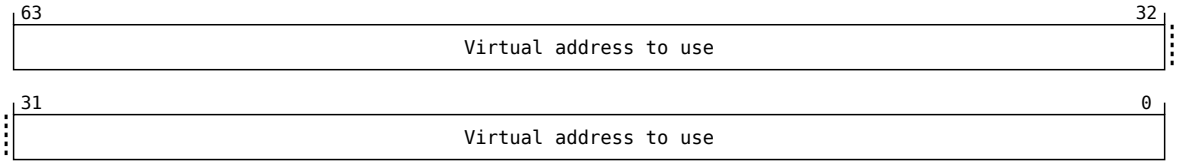
Configuration

There are no configuration notes.

Attributes

DC CIVAC is a 64-bit System instruction.

Field descriptions



VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC CIVAC

If EL0 access is enabled, when executed at EL0, the instruction may generate a Permission fault, subject to the constraints described in 'MMU faults generated by cache maintenance operations'.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see 'The data cache maintenance instruction (DC)'.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`DC CIVAC, <Xt>`

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1110	0b001

```
if PSTATE.EL == EL0 then
    if !AArch64.CanTrapDC(CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
    elsif !ELIsInHost(EL0) && SCTLR_EL1.UCI == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif !ELIsInHost(EL0) && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_PoC);
elsif PSTATE.EL == EL1 then
    if !AArch64.CanTrapDC(CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_PoC) then
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        ExecuteAsNOP();
    elsif HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_PoC);
elsif PSTATE.EL == EL2 then
    if !AArch64.CanTrapDC(CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
    else
        if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Data, CacheOp_CleanInvalidate, CacheOpScope_PoC);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.3 DC CSW, Data or unified Cache line Clean by Set/Way

The DC CSW characteristics are:

Purpose

Clean data cache by set/way.

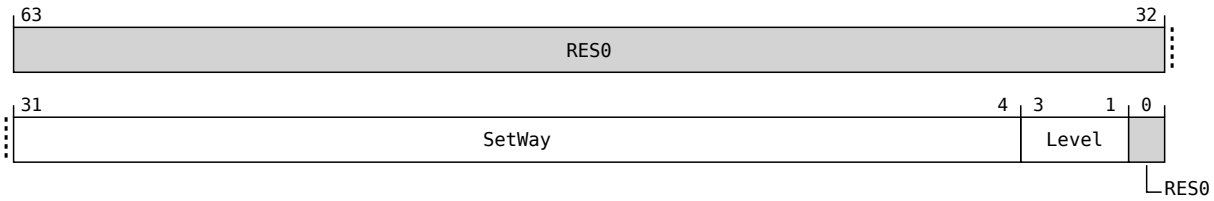
Configuration

There are no configuration notes.

Attributes

DC CSW is a 64-bit System instruction.

Field descriptions



Bits [63:32]

Reserved, RES0.

SetWay, bits [31:4]

Contains two fields:

- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

$A = Log_2(ASSOCIATIVITY)$, $L = Log_2(LINELEN)$, $B = (L + S)$, $S = Log_2(NSETS)$.

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

Level, bits [3:1]

Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

Bit [0]

Reserved, RES0.

Executing DC CSW

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED.
- The instruction performs cache maintenance on one of:
  - No cache lines.
  - A single arbitrary cache line.
  - Multiple arbitrary cache lines.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC CSW, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b1010	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TSW == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Clean, CacheOpScope_SetWay);
elsif PSTATE.EL == EL2 then
    AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Clean, CacheOpScope_SetWay);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.4 DC CVAC, Data or unified Cache line Clean by VA to PoC

The DC CVAC characteristics are:

Purpose

Clean data cache by address to Point of Coherency.

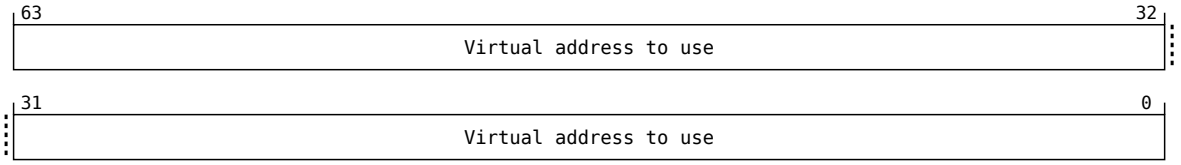
Configuration

There are no configuration notes.

Attributes

DC CVAC is a 64-bit System instruction.

Field descriptions



VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC CVAC

If EL0 access is enabled, when executed at EL0, the instruction may generate a Permission fault, subject to the constraints described in 'MMU faults generated by cache maintenance operations'.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see 'The data cache maintenance instruction (DC)'.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`DC CVAC, <Xt>`

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1010	0b001

```
if PSTATE.EL == EL0 then
    if !AArch64.CanTrapDC(CacheType_Data, CacheOp_Clean, CacheOpScope_PoC) then
        ExecuteAsNOP();
    elsif !ELIsInHost(EL0) && SCTLR_EL1.UCI == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif !ELIsInHost(EL0) && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Clean, CacheOpScope_PoC);
elsif PSTATE.EL == EL1 then
    if !AArch64.CanTrapDC(CacheType_Data, CacheOp_Clean, CacheOpScope_PoC) then
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        ExecuteAsNOP();
    elsif HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Clean, CacheOpScope_PoC);
elsif PSTATE.EL == EL2 then
    if !AArch64.CanTrapDC(CacheType_Data, CacheOp_Clean, CacheOpScope_PoC) then
        ExecuteAsNOP();
    else
        if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Clean, CacheOpScope_PoC);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.5 DC CVADP, Data or unified Cache line Clean by VA to PoDP

The DC CVADP characteristics are:

Purpose

Clean data cache by address to Point of Deep Persistence.

If the memory system does not identify a Point of Deep Persistence, then this instruction behaves as a DC CVAP.

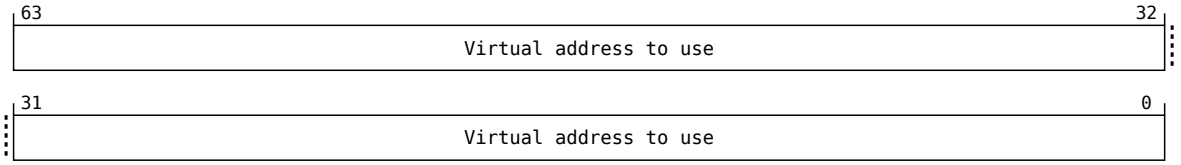
Configuration

This system instruction is present only when FEAT_DPB2 is implemented. Otherwise, direct accesses to DC CVADP are UNDEFINED.

Attributes

DC CVADP is a 64-bit System instruction.

Field descriptions



VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC CVADP

If EL0 access is enabled, when executed at EL0, the instruction may generate a Permission fault, subject to the constraints described in 'MMU faults generated by cache maintenance operations'.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see 'The data cache maintenance instruction (DC)'.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`DC CVADP, <Xt>`

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1101	0b001

```
if !IsFeatureImplemented(FEAT_DPB2) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if !AArch64.CanTrapDC(CacheType_Data, CacheOp_Clean, CacheOpScope_PoDP) then
        ExecuteAsNOP();
    elsif !ELIsInHost(EL0) && SCTLR_EL1.UCI == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif !ELIsInHost(EL0) && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoDP) then
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Clean, CacheOpScope_PoDP);
elsif PSTATE.EL == EL1 then
    if !AArch64.CanTrapDC(CacheType_Data, CacheOp_Clean, CacheOpScope_PoDP) then
        ExecuteAsNOP();
    elsif HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoDP) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Clean, CacheOpScope_PoDP);
elsif PSTATE.EL == EL2 then
    if !AArch64.CanTrapDC(CacheType_Data, CacheOp_Clean, CacheOpScope_PoDP) then
        ExecuteAsNOP();
    else
        if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoDP) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Clean, CacheOpScope_PoDP);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.6 DC CVAP, Data or unified Cache line Clean by VA to PoP

The DC CVAP characteristics are:

Purpose

Clean data cache by address to Point of Persistence.

If the memory system does not identify a Point of Persistence, then this instruction behaves as a DC CVAC.

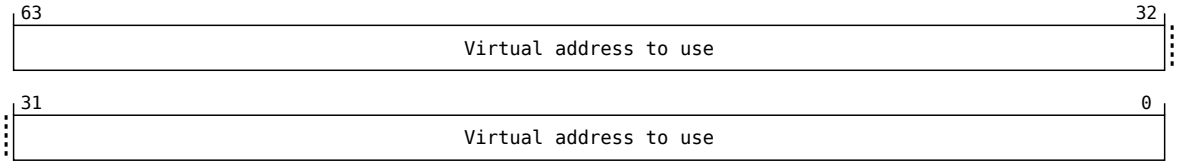
Configuration

This system instruction is present only when FEAT_DPB is implemented. Otherwise, direct accesses to DC CVAP are UNDEFINED.

Attributes

DC CVAP is a 64-bit System instruction.

Field descriptions



VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC CVAP

If EL0 access is enabled, when executed at EL0, the instruction may generate a Permission fault, subject to the constraints described in 'MMU faults generated by cache maintenance operations'.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see 'The data cache maintenance instruction (DC)'.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`DC CVAP, <Xt>`

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1100	0b001

```
if !IsFeatureImplemented(FEAT_DPB) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if !AArch64.CanTrapDC(CacheType_Data, CacheOp_Clean, CacheOpScope_PoP) then
        ExecuteAsNOP();
    elsif !ELIsInHost(EL0) && SCTLR_EL1.UCI == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif !ELIsInHost(EL0) && HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoP) then
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Clean, CacheOpScope_PoP);
elsif PSTATE.EL == EL1 then
    if !AArch64.CanTrapDC(CacheType_Data, CacheOp_Clean, CacheOpScope_PoP) then
        ExecuteAsNOP();
    elsif HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoP) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Clean, CacheOpScope_PoP);
elsif PSTATE.EL == EL2 then
    if !AArch64.CanTrapDC(CacheType_Data, CacheOp_Clean, CacheOpScope_PoP) then
        ExecuteAsNOP();
    else
        if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoP) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Clean, CacheOpScope_PoP);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.7 DC CVAU, Data or unified Cache line Clean by VA to PoU

The DC CVAU characteristics are:

Purpose

Clean data cache by address to Point of Unification.

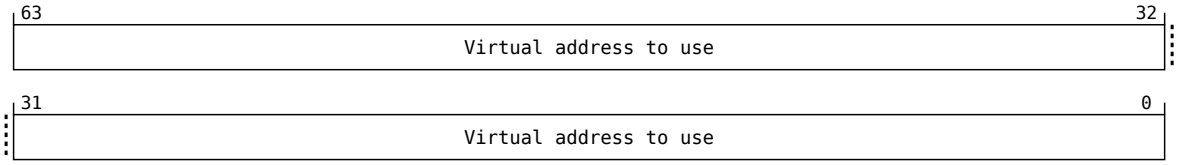
Configuration

There are no configuration notes.

Attributes

DC CVAU is a 64-bit System instruction.

Field descriptions



VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC CVAU

If EL0 access is enabled, when executed at EL0, the instruction may generate a Permission fault, subject to the constraints described in 'MMU faults generated by cache maintenance operations'.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see 'The data cache maintenance instruction (DC)'.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`DC CVAU, <Xt>`

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b1011	0b001

```
if PSTATE.EL == EL0 then
    if !AArch64.CanTrapDC(CacheType_Data, CacheOp_Clean, CacheOpScope_PoU) then
        ExecuteAsNOP();
    elsif !ELIsInHost(EL0) && SCTLR_EL1.UCI == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif !ELIsInHost(EL0) && HCR_EL2.TPU == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoU) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Clean, CacheOpScope_PoU);
elsif PSTATE.EL == EL1 then
    if !AArch64.CanTrapDC(CacheType_Data, CacheOp_Clean, CacheOpScope_PoU) then
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        ExecuteAsNOP();
    elsif HCR_EL2.TPU == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoU) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Clean, CacheOpScope_PoU);
elsif PSTATE.EL == EL2 then
    if !AArch64.CanTrapDC(CacheType_Data, CacheOp_Clean, CacheOpScope_PoU) then
        ExecuteAsNOP();
    else
        if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_Clean, CacheOpScope_PoU) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Clean, CacheOpScope_PoU);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.8 DC ISW, Data or unified Cache line Invalidate by Set/Way

The DC ISW characteristics are:

Purpose

Invalidate data cache by set/way.

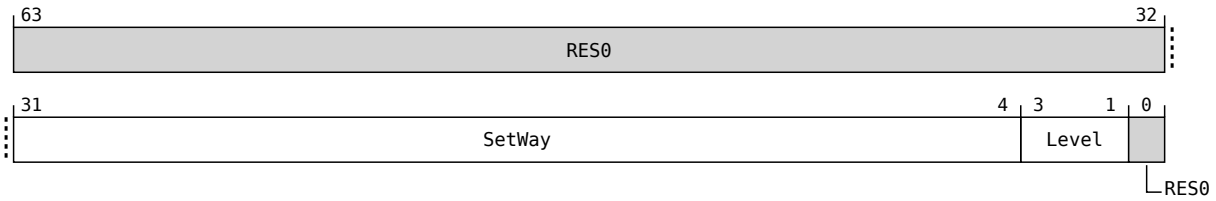
Configuration

There are no configuration notes.

Attributes

DC ISW is a 64-bit System instruction.

Field descriptions



Bits [63:32]

Reserved, RES0.

SetWay, bits [31:4]

Contains two fields:

- Way, bits[31:32-A], the number of the way to operate on.
- Set, bits[B-1:L], the number of the set to operate on.

Bits[L-1:4] are RES0.

$A = Log_2(ASSOCIATIVITY)$, $L = Log_2(LINELEN)$, $B = (L + S)$, $S = Log_2(NSETS)$.

ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

Level, bits [3:1]

Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

Bit [0]

Reserved, RES0.

Executing DC ISW

If this instruction is executed with a set, way or level argument that is larger than the value supported by the implementation then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED.
- The instruction performs cache maintenance on one of:
  - No cache lines.
  - A single arbitrary cache line.
  - Multiple arbitrary cache lines.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`DC ISW, <Xt>`

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b0110	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TSW == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Invalidate, CacheOpScope_SetWay);
elsif PSTATE.EL == EL2 then
    AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Invalidate, CacheOpScope_SetWay);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.9 DC IVAC, Data or unified Cache line Invalidate by VA to PoC

The DC IVAC characteristics are:

Purpose

Invalidate data cache by address to Point of Coherency.

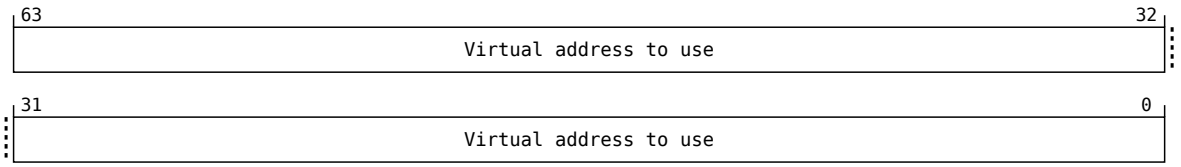
Configuration

There are no configuration notes.

Attributes

DC IVAC is a 64-bit System instruction.

Field descriptions



VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing DC IVAC

When the instruction is executed, it can generate a watchpoint, which is prioritized in the same way as other watchpoints. If a watchpoint is generated, the CM bit in the ESR_ELx.ISS field is set to 1.

This instruction requires write access permission to the VA, otherwise it generates a Permission fault, subject to the constraints described in 'MMU faults generated by cache maintenance operations'.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see 'The data cache maintenance instruction (DC)'.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC IVAC, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b0110	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if !AArch64.CanTrapDC(CacheType_Data, CacheOp_Invalidate, CacheOpScope_PoC) then
        ExecuteAsNOP();
    elsif HCR_EL2.TPCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_Invalidate, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Invalidate, CacheOpScope_PoC);
elsif PSTATE.EL == EL2 then
    if !AArch64.CanTrapDC(CacheType_Data, CacheOp_Invalidate, CacheOpScope_PoC) then
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        ExecuteAsNOP();
    else
        if AArch64.TreatDCAsNOP(CacheType_Data, CacheOp_Invalidate, CacheOpScope_PoC) then
            ExecuteAsNOP();
        else
            AArch64.DC(X[t, 64], CacheType_Data, CacheOp_Invalidate, CacheOpScope_PoC);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.10 DC ZVA, Data Cache Zero by VA

The DC ZVA characteristics are:

Purpose

Zero data cache by address. Zeroes a naturally aligned block of N bytes, where the size of N is identified in DCZID_EL0.

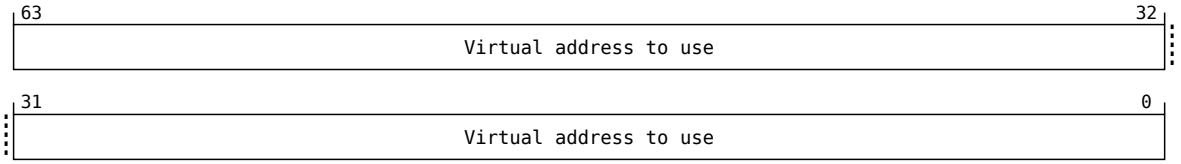
Configuration

There are no configuration notes.

Attributes

DC ZVA is a 64-bit System instruction.

Field descriptions



VA, bits [63:0]

Virtual address to use. There is no alignment restriction on the address within the block of N bytes that is used.

Executing DC ZVA

When this instruction is executed, it can generate memory faults or watchpoints which are prioritized in the same way as other memory-related faults or watchpoints. If a synchronous Data Abort fault or a watchpoint is generated, the CM bit in the ESR_ELx.ISS field is set to 0.

If the memory region being zeroed is any type of Device memory, this instruction generates an Alignment fault which is prioritized in the same way as other Alignment faults that are determined by the memory type.

This instruction applies to Normal memory regardless of cacheability attributes.

This instruction behaves as a set of Stores to each byte within the block being accessed, and so it:

- Generates a Permission fault if the translation system does not permit writes to the locations.
- Requires the same considerations for ordering and the management of coherency as any other store instructions.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
DC ZVA, <Xt>
```

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b0100	0b001

```
if PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && SCTLR_EL1.DZE == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif !ELIsInHost(EL0) && HCR_EL2.TDZ == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.MemZero(X[t, 64], CacheType_Data);
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TDZ == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.MemZero(X[t, 64], CacheType_Data);
elsif PSTATE.EL == EL2 then
    AArch64.MemZero(X[t, 64], CacheType_Data);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.11 IC IALLU, Instruction Cache Invalidate All to PoU

The IC IALLU characteristics are:

Purpose

Invalidate all instruction caches of the PE executing the instruction to the Point of Unification.

Configuration

There are no configuration notes.

Attributes

IC IALLU is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing IC IALLU

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
IC IALLU{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b0101	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if !AArch64.CanTrapIC(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) then
        ExecuteAsNOP();
    elsif HCR_EL2.TPU == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HCR_EL2.FB == '1' then
        if AArch64.TreatICAsNOP(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) then
            ExecuteAsNOP();
        else
            AArch64.IC(CacheOpScope_ALLUIS);
    else
        if AArch64.TreatICAsNOP(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) then
            ExecuteAsNOP();
        else
            AArch64.IC(CacheOpScope_ALLU);
elsif PSTATE.EL == EL2 then
    if !AArch64.CanTrapIC(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) then
        ExecuteAsNOP();
    else
        if AArch64.TreatICAsNOP(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) then
            ExecuteAsNOP();
        else
            AArch64.IC(CacheOpScope_ALLU);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.12 IC IALLUIS, Instruction Cache Invalidate All to PoU, Inner Shareable

The IC IALLUIS characteristics are:

Purpose

Invalidate all instruction caches in the Inner Shareable domain of the PE executing the instruction to the Point of Unification.

Configuration

There are no configuration notes.

Attributes

IC IALLUIS is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing IC IALLUIS

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
IC IALLUIS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if !AArch64.CanTrapIC(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) then
        ExecuteAsNOP();
    elsif HCR_EL2.TPU == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if AArch64.TreatICAsNOP(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) then
            ExecuteAsNOP();
        else
            AArch64.IC(CacheOpScope_ALLUIS);
elsif PSTATE.EL == EL2 then
    if !AArch64.CanTrapIC(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) then
        ExecuteAsNOP();
    else
        if AArch64.TreatICAsNOP(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) then
            ExecuteAsNOP();
        else
            AArch64.IC(CacheOpScope_ALLUIS);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.3.13 IC IVAU, Instruction Cache line Invalidate by VA to PoU

The IC IVAU characteristics are:

Purpose

Invalidate instruction cache by address to Point of Unification.

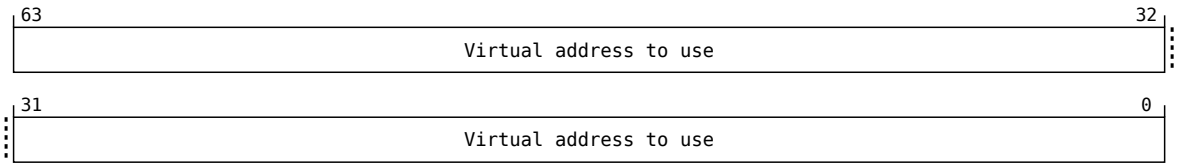
Configuration

There are no configuration notes.

Attributes

IC IVAU is a 64-bit System instruction.

Field descriptions



VA, bits [63:0]

Virtual address to use. No alignment restrictions apply to this VA.

Executing IC IVAU

If EL0 access is enabled, when executed at EL0, the instruction may generate a Permission fault, subject to the constraints described in 'MMU faults generated by cache maintenance operations'.

Execution of this instruction might require an address translation from VA to PA, and that translation might fault. For more information, see 'The instruction cache maintenance instruction (IC)'.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`IC IVAU{, <Xt>}`

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b0101	0b001

```
if PSTATE.EL == EL0 then
    if !AArch64.CanTrapIC(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) then
        ExecuteAsNOP();
    elsif !ELIsInHost(EL0) && SCTLR_EL1.UCI == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif !ELIsInHost(EL0) && HCR_EL2.TPU == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if AArch64.TreatICAsNOP(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) then
            ExecuteAsNOP();
        else
            AArch64.IC(X[t, 64], CacheOpScope_PoU);
elsif PSTATE.EL == EL1 then
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if !AArch64.CanTrapIC(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) then
        ExecuteAsNOP();
    elsif HCR_EL2.TPU == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if AArch64.TreatICAsNOP(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) then
            ExecuteAsNOP();
        else
            AArch64.IC(X[t, 64], CacheOpScope_PoU);
elsif PSTATE.EL == EL2 then
    if !AArch64.CanTrapIC(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) then
        ExecuteAsNOP();
    else
        if AArch64.TreatICAsNOP(CacheType_Instruction, CacheOp_Invalidate, CacheOpScope_PoU) then
            ExecuteAsNOP();
        else
            AArch64.IC(X[t, 64], CacheOpScope_PoU);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4 A64 System instructions for address translation

This section lists the A64 System instructions for address translation.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4.1 AT S12E0R, Address Translate Stages 1 and 2 EL0 Read

The AT S12E0R characteristics are:

Purpose

Performs stage 1 and 2 address translations from EL0, with permissions as if reading from the given virtual address from EL0, using the EL1&0 translation regime.

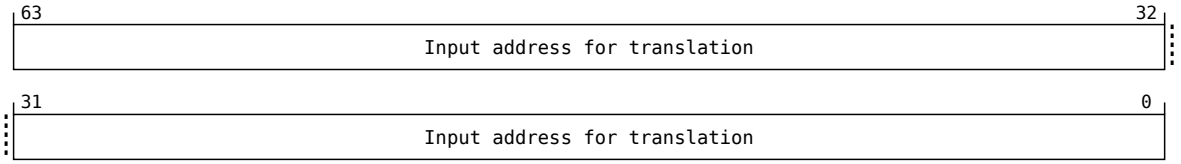
Configuration

There are no configuration notes.

Attributes

AT S12E0R is a 64-bit System instruction.

Field descriptions



IA, bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

Executing AT S12E0R

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`AT S12E0R, <Xt>`

op0	op1	CRn	CRm	op2
0b01	0b100	0b0111	0b1000	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.<DC,VM> == '00' then
        AArch64.AT(X[t, 64], TranslationStage_1, EL0, ATAccess_Read);
    else
        AArch64.AT(X[t, 64], TranslationStage_12, EL0, ATAccess_Read);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4.2 AT S12E0W, Address Translate Stages 1 and 2 EL0 Write

The AT S12E0W characteristics are:

Purpose

Performs stage 1 and 2 address translations from EL0, with permissions as if writing to the given virtual address from EL0, using the EL1&0 translation regime.

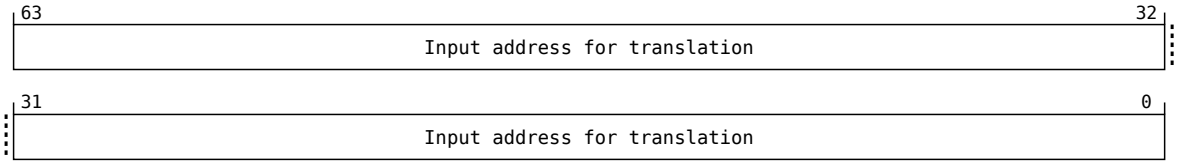
Configuration

There are no configuration notes.

Attributes

AT S12E0W is a 64-bit System instruction.

Field descriptions



IA, bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

Executing AT S12E0W

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`AT S12E0W, <Xt>`

op0	op1	CRn	CRm	op2
0b01	0b100	0b0111	0b1000	0b111

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.<DC,VM> == '00' then
        AArch64.AT(X[t, 64], TranslationStage_1, EL0, ATAccess_Write);
    else
        AArch64.AT(X[t, 64], TranslationStage_12, EL0, ATAccess_Write);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4.3 AT S12E1R, Address Translate Stages 1 and 2 EL1 Read

The AT S12E1R characteristics are:

Purpose

Performs stage 1 and 2 address translation, with permissions as if reading from the given virtual address from EL1, using the EL1&0 translation regime.

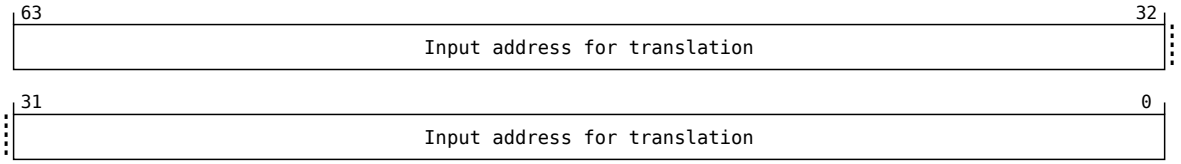
Configuration

There are no configuration notes.

Attributes

AT S12E1R is a 64-bit System instruction.

Field descriptions



IA, bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

Executing AT S12E1R

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`AT S12E1R, <Xt>`

op0	op1	CRn	CRm	op2
0b01	0b100	0b0111	0b1000	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.<DC,VM> == '00' then
        AArch64.AT(X[t, 64], TranslationStage_1, EL1, ATAccess_Read);
    else
        AArch64.AT(X[t, 64], TranslationStage_12, EL1, ATAccess_Read);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4.4 AT S12E1W, Address Translate Stages 1 and 2 EL1 Write

The AT S12E1W characteristics are:

Purpose

Performs stage 1 and 2 address translation, with permissions as if writing to the given virtual address from EL1, using the EL1&0 translation regime.

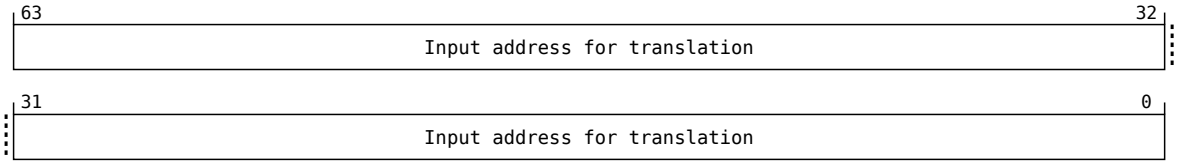
Configuration

There are no configuration notes.

Attributes

AT S12E1W is a 64-bit System instruction.

Field descriptions



IA, bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

Executing AT S12E1W

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`AT S12E1W, <Xt>`

op0	op1	CRn	CRm	op2
0b01	0b100	0b0111	0b1000	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.<DC,VM> == '00' then
        AArch64.AT(X[t, 64], TranslationStage_1, EL1, ATAccess_Write);
    else
        AArch64.AT(X[t, 64], TranslationStage_12, EL1, ATAccess_Write);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4.5 AT S1E0R, Address Translate Stage 1 EL0 Read

The AT S1E0R characteristics are:

Purpose

Performs stage 1 address translation from EL0, with permissions as if reading from the given virtual address from EL0, using the EL1&0 translation regime.

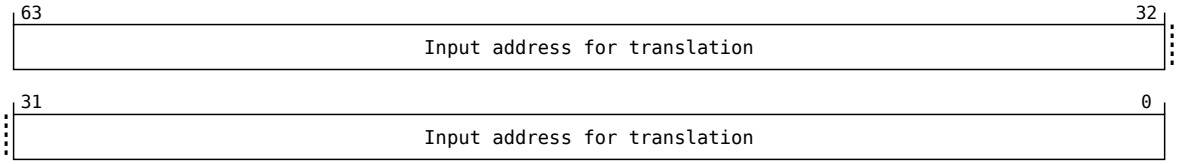
Configuration

There are no configuration notes.

Attributes

AT S1E0R is a 64-bit System instruction.

Field descriptions



IA, bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

Executing AT S1E0R

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`AT S1E0R, <Xt>`

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b1000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    AArch64.AT(X[t, 64], TranslationStage_1, EL0, ATAccess_Read);
elsif PSTATE.EL == EL2 then
    AArch64.AT(X[t, 64], TranslationStage_1, EL0, ATAccess_Read);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4.6 AT S1E0W, Address Translate Stage 1 EL0 Write

The AT S1E0W characteristics are:

Purpose

Performs stage 1 address translation from EL0, with permissions as if writing to the given virtual address from EL0, using the EL1&0 translation regime.

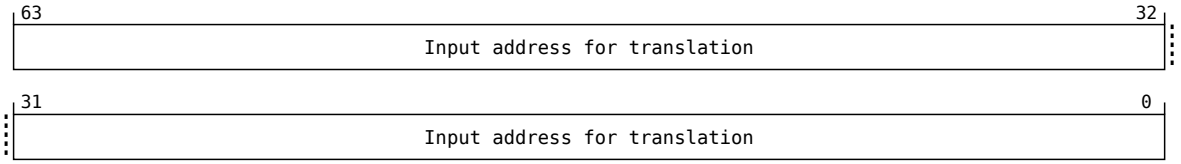
Configuration

There are no configuration notes.

Attributes

AT S1E0W is a 64-bit System instruction.

Field descriptions



IA, bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

Executing AT S1E0W

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`AT S1E0W, <Xt>`

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b1000	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    AArch64.AT(X[t, 64], TranslationStage_1, EL0, ATAccess_Write);
elsif PSTATE.EL == EL2 then
    AArch64.AT(X[t, 64], TranslationStage_1, EL0, ATAccess_Write);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4.7 AT S1E1R, Address Translate Stage 1 EL1 Read

The AT S1E1R characteristics are:

Purpose

Performs stage 1 address translation, with permissions as if reading from the given virtual address from EL1, using the EL1&0 translation regime.

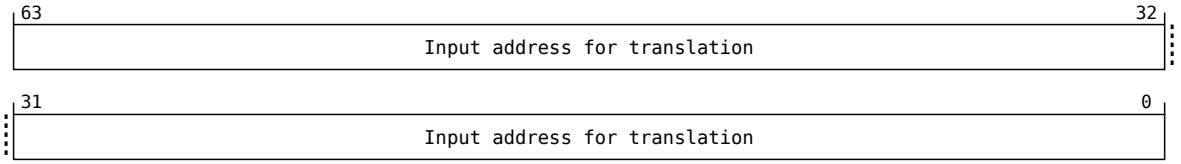
Configuration

There are no configuration notes.

Attributes

AT S1E1R is a 64-bit System instruction.

Field descriptions



IA, bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

Executing AT S1E1R

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`AT S1E1R, <Xt>`

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b1000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    AArch64.AT(X[t, 64], TranslationStage_1, EL1, ATAccess_Read);
elsif PSTATE.EL == EL2 then
    AArch64.AT(X[t, 64], TranslationStage_1, EL1, ATAccess_Read);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4.8 AT S1E1RP, Address Translate Stage 1 EL1 Read PAN

The AT S1E1RP characteristics are:

Purpose

Performs a stage 1 address translation, where the value of PSTATE.PAN determines if a read from a location will generate a Permission fault for a privileged access, using the EL1&0 translation regime, accessed from EL1.

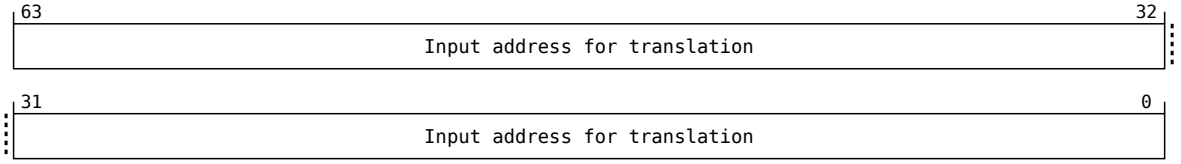
Configuration

This system instruction is present only when FEAT_PAN2 is implemented. Otherwise, direct accesses to AT S1E1RP are UNDEFINED.

Attributes

AT S1E1RP is a 64-bit System instruction.

Field descriptions



IA, bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

Executing AT S1E1RP

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`AT S1E1RP, <Xt>`

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b1001	0b000

```
if !IsFeatureImplemented(FEAT_PAN2) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    AArch64.AT(X[t, 64], TranslationStage_1, EL1, ATAccess_ReadPAN);
elsif PSTATE.EL == EL2 then
    AArch64.AT(X[t, 64], TranslationStage_1, EL1, ATAccess_ReadPAN);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4.9 AT S1E1W, Address Translate Stage 1 EL1 Write

The AT S1E1W characteristics are:

Purpose

Performs stage 1 address translation, with permissions as if writing to the given virtual address from EL1, using the EL1&0 translation regime.

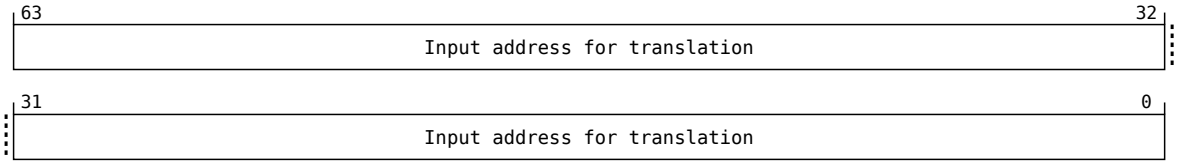
Configuration

There are no configuration notes.

Attributes

AT S1E1W is a 64-bit System instruction.

Field descriptions



IA, bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

Executing AT S1E1W

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`AT S1E1W, <Xt>`

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b1000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    AArch64.AT(X[t, 64], TranslationStage_1, EL1, ATAccess_Write);
elsif PSTATE.EL == EL2 then
    AArch64.AT(X[t, 64], TranslationStage_1, EL1, ATAccess_Write);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4.10 AT S1E1WP, Address Translate Stage 1 EL1 Write PAN

The AT S1E1WP characteristics are:

Purpose

Performs a stage 1 address translation, where the value of PSTATE.PAN determines if a write to a location will generate a Permission fault for a privileged access, using the EL1&0 translation regime, accessed from EL1.

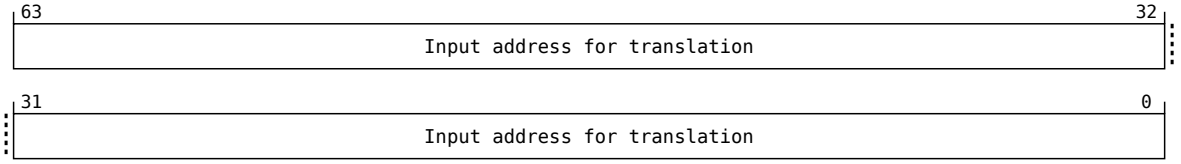
Configuration

This system instruction is present only when FEAT_PAN2 is implemented. Otherwise, direct accesses to AT S1E1WP are UNDEFINED.

Attributes

AT S1E1WP is a 64-bit System instruction.

Field descriptions



IA, bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

Executing AT S1E1WP

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`AT S1E1WP, <Xt>`

op0	op1	CRn	CRm	op2
0b01	0b000	0b0111	0b1001	0b001

```
if !IsFeatureImplemented(FEAT_PAN2) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    AArch64.AT(X[t, 64], TranslationStage_1, EL1, ATAccess_WritePAN);
elsif PSTATE.EL == EL2 then
    AArch64.AT(X[t, 64], TranslationStage_1, EL1, ATAccess_WritePAN);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4.11 AT S1E2R, Address Translate Stage 1 EL2 Read

The AT S1E2R characteristics are:

Purpose

Performs stage 1 address translation as defined for EL2, with permissions as if reading from the given virtual address.

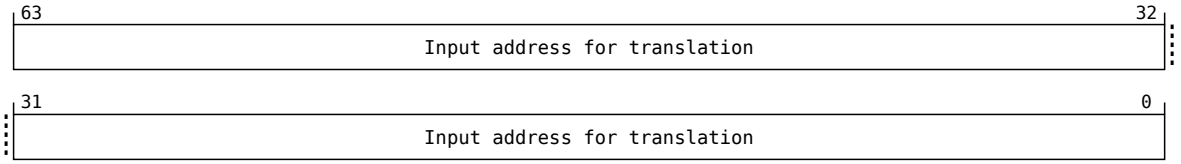
Configuration

There are no configuration notes.

Attributes

AT S1E2R is a 64-bit System instruction.

Field descriptions



IA, bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

Executing AT S1E2R

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`AT S1E2R, <Xt>`

op0	op1	CRn	CRm	op2
0b01	0b100	0b0111	0b1000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.AT(X[t, 64], TranslationStage_1, EL2, ATAccess_Read);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4.12 AT S1E2W, Address Translate Stage 1 EL2 Write

The AT S1E2W characteristics are:

Purpose

Performs stage 1 address translation as defined for EL2, with permissions as if writing to the given virtual address.

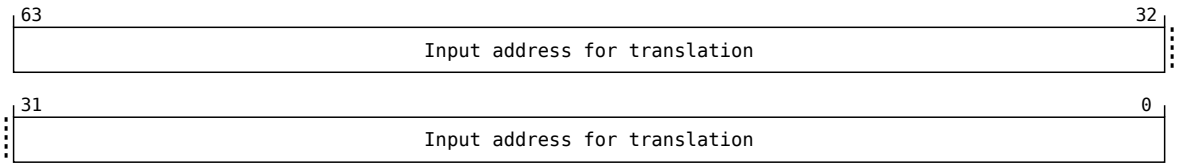
Configuration

There are no configuration notes.

Attributes

AT S1E2W is a 64-bit System instruction.

Field descriptions



IA, bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

Executing AT S1E2W

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`AT S1E2W, <Xt>`

op0	op1	CRn	CRm	op2
0b01	0b100	0b0111	0b1000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.AT(X[t, 64], TranslationStage_1, EL2, ATAccess_Write);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4.13 AT S1E3R, Address Translate Stage 1 EL3 Read

The AT S1E3R characteristics are:

Purpose

Performs stage 1 address translation as defined for EL3, with permissions as if reading from the given virtual address.

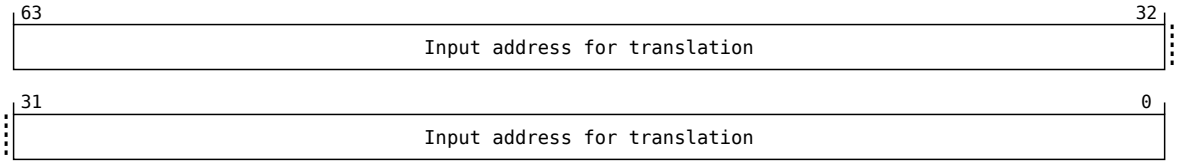
Configuration

There are no configuration notes.

Attributes

AT S1E3R is a 64-bit System instruction.

Field descriptions



IA, bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

Executing AT S1E3R

This system instruction is an alias of the SYS instruction.

This system instruction encoding is Undefined()

Accesses to this instruction use the following encodings in the System instruction encoding space:

`AT S1E3R, <Xt>`

op0	op1	CRn	CRm	op2
0b01	0b110	0b0111	0b1000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.4.14 AT S1E3W, Address Translate Stage 1 EL3 Write

The AT S1E3W characteristics are:

Purpose

Performs stage 1 address translation as defined for EL3, with permissions as if writing to the given virtual address.

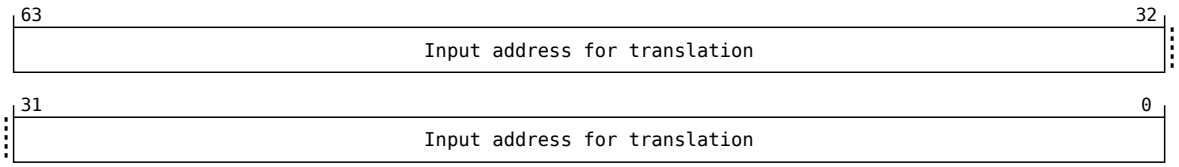
Configuration

There are no configuration notes.

Attributes

AT S1E3W is a 64-bit System instruction.

Field descriptions



IA, bits [63:0]

Input address for translation. The resulting address can be read from the PAR_EL1.

Executing AT S1E3W

This system instruction is an alias of the SYS instruction.

This system instruction encoding is Undefined()

Accesses to this instruction use the following encodings in the System instruction encoding space:

`AT S1E3W, <Xt>`

op0	op1	CRn	CRm	op2
0b01	0b110	0b0111	0b1000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5 A64 System instructions for TLB maintenance

This section lists the A64 System instructions for TLB maintenance.

For more information about these instructions see TLB maintenance instructions. In particular, for the full description of the scope of each instruction see TLB maintenance instruction scope.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.1 TLBI ALLE1, TLB Invalidate All, EL1

The TLBI ALLE1 characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 or stage 2 translation table entry, from any level of the translation table walk.
- The entry would be required to translate an address using the EL1&0 translation regime.

The invalidation applies to entries with any VMID.

The invalidation only applies to the PE that executes this System instruction.

Note

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

Configuration

There are no configuration notes.

Attributes

TLBI ALLE1 is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing TLBI ALLE1

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI ALLE1{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0111	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, Broadcast_NSH, TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.2 TLBI ALLE1IS, TLB Invalidate All, EL1, Inner Shareable

The TLBI ALLE1IS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 or stage 2 translation table entry, from any level of the translation table walk.
- The entry would be required to translate an address using the EL1&0 translation regime.

The invalidation applies to entries with any VMID.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

Configuration

There are no configuration notes.

Attributes

TLBI ALLE1IS is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing TLBI ALLE1IS

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI ALLE1IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0011	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, Broadcast_ISH, TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.3 TLBI ALLE1OS, TLB Invalidate All, EL1, Outer Shareable

The TLBI ALLE1OS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 or stage 2 translation table entry, from any level of the translation table walk.
- The entry would be required to translate an address using the EL1&0 translation regime.

The invalidation applies to entries with any VMID.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

Configuration

This system instruction is present only when FEAT_TLBIOS is implemented. Otherwise, direct accesses to TLBI ALLE1OS are UNDEFINED.

Attributes

TLBI ALLE1OS is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing TLBI ALLE1OS

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI ALLE1OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0001	0b100

```
if !IsFeatureImplemented(FEAT_TLBIOS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_ALL(SecurityStateAtEL(EL1), Regime_EL10, Broadcast_OSH, TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.4 TLBI ALLE2, TLB Invalidate All, EL2

The TLBI ALLE2 characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- The entry would be required to translate any address using the EL2&0 or EL2 translation regime.

The invalidation only applies to the PE that executes this System instruction.

Configuration

There are no configuration notes.

Attributes

TLBI ALLE2 is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing TLBI ALLE2

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI ALLE2{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0111	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL2, Broadcast_NSH, TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.5 TLBI ALLE2IS, TLB Invalidate All, EL2, Inner Shareable

The TLBI ALLE2IS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- The entry would be required to translate any address using the EL2&0 or EL2 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Configuration

There are no configuration notes.

Attributes

TLBI ALLE2IS is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing TLBI ALLE2IS

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI ALLE2IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0011	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL2, Broadcast_ISH, TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.6 TLBI ALLE2OS, TLB Invalidate All, EL2, Outer Shareable

The TLBI ALLE2OS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- The entry would be required to translate any address using the EL2&0 or EL2 translation regime.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Configuration

This system instruction is present only when FEAT_TLBIOS is implemented. Otherwise, direct accesses to TLBI ALLE2OS are UNDEFINED.

Attributes

TLBI ALLE2OS is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing TLBI ALLE2OS

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI ALLE2OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_TLBIOS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_ALL(SecurityStateAtEL(EL2), Regime_EL2, Broadcast_OSH, TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.7 TLBI ALLE3, TLB Invalidate All, EL3

The TLBI ALLE3 characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- The entry would be required to translate an address using the EL3 translation regime.

The invalidation applies to the PE that executes this System instruction.

Configuration

There are no configuration notes.

Attributes

TLBI ALLE3 is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing TLBI ALLE3

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

This system instruction encoding is Undefined()

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI ALLE3{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0111	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.8 TLBI ALLE3IS, TLB Invalidate All, EL3, Inner Shareable

The TLBI ALLE3IS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- The entry would be required to translate an address using the EL3 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Configuration

There are no configuration notes.

Attributes

TLBI ALLE3IS is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing TLBI ALLE3IS

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

This system instruction encoding is Undefined()

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI ALLE3IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0011	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.9 TLBI ALLE3OS, TLB Invalidate All, EL3, Outer Shareable

The TLBI ALLE3OS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- The entry would be required to translate an address using the EL3 translation regime.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Configuration

This system instruction is present only when FEAT_TLBIOS is implemented. Otherwise, direct accesses to TLBI ALLE3OS are UNDEFINED.

Attributes

TLBI ALLE3OS is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing TLBI ALLE3OS

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

This system instruction encoding is Undefined()

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI ALLE3OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_TLBIOS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.10 TLBI ASIDE1, TLB Invalidate by ASID, EL1

The TLBI ASIDE1 characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used for the specified ASID, and either:
  - Is from a level of lookup above the final level.
  - Is a non-global entry from the final level of lookup.

The entry would be used with the current VMID and would be required to translate an address using the EL1&0 translation regime for the Security state.

The invalidation applies to the PE that executes this System instruction.

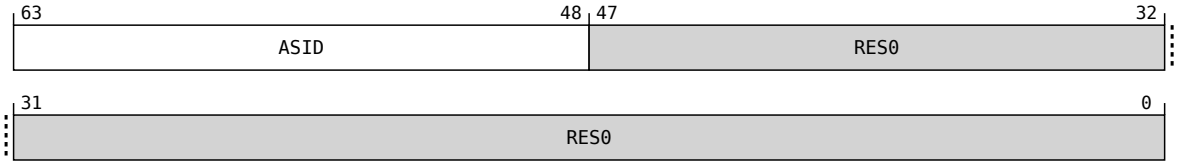
Configuration

There are no configuration notes.

Attributes

TLBI ASIDE1 is a 64-bit System instruction.

Field descriptions



ASID, bits [63:48]

ASID value to match. Any appropriate TLB entries that match the ASID values will be affected by this System instruction.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

Bits [47:0]

Reserved, RES0.

Executing TLBI ASIDE1

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI ASIDE1{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0111	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TTLB == '1' then
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HCR_EL2.FB == '1' then
        AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
            ↪TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBI_AllAttr, X[t,
            ↪64]);
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBI_AllAttr, X[t,
        ↪64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.11 TLBI ASIDE1IS, TLB Invalidate by ASID, EL1, Inner Shareable

The TLBI ASIDE1IS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used for the specified ASID, and either:
  - Is from a level of lookup above the final level.
  - Is a non-global entry from the final level of lookup.

The entry would be used with the current VMID and would be required to translate an address using the EL1&0 translation regime for the Security state.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

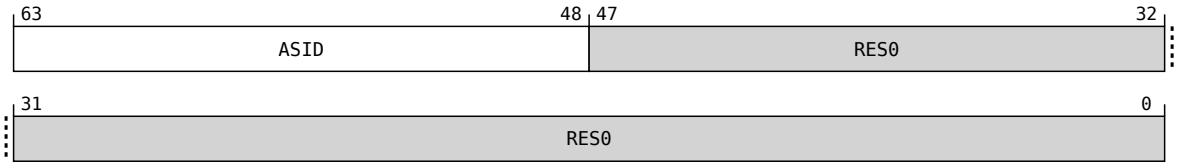
Configuration

There are no configuration notes.

Attributes

TLBI ASIDE1IS is a 64-bit System instruction.

Field descriptions



ASID, bits [63:48]

ASID value to match. Any appropriate TLB entries that match the ASID values will be affected by this System instruction.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

Bits [47:0]

Reserved, RES0.

Executing TLBI ASIDE1IS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`TLBI ASIDE1IS{, <Xt>}`

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0011	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBI_AllAttr, X[t,
            ↪64]);
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBI_AllAttr, X[t,
        ↪64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.12 TLBI ASIDE1OS, TLB Invalidate by ASID, EL1, Outer Shareable

The TLBI ASIDE1OS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used for the specified ASID, and either:
  - Is from a level of lookup above the final level.
  - Is a non-global entry from the final level of lookup.

The entry would be used with the current VMID and would be required to translate an address using the EL1&0 translation regime for the Security state.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

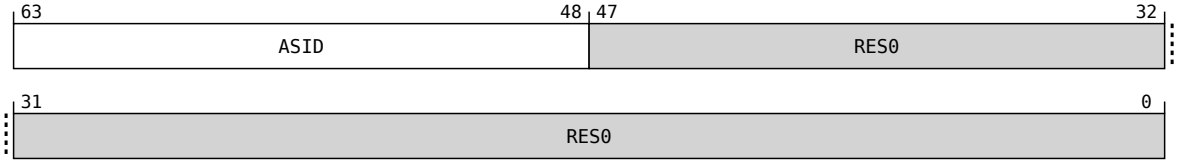
Configuration

This system instruction is present only when FEAT_TLBIOS is implemented. Otherwise, direct accesses to TLBI ASIDE1OS are UNDEFINED.

Attributes

TLBI ASIDE1OS is a 64-bit System instruction.

Field descriptions



ASID, bits [63:48]

ASID value to match. Any appropriate TLB entries that match the ASID values will be affected by this System instruction.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

Bits [47:0]

Reserved, RES0.

Executing TLBI ASIDE1OS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI ASIDE1OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0001	0b010

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_TLBIOS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBI_AllAttr, X[t,
            ↪64]);
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBI_AllAttr, X[t,
        ↪64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.13 TLBI IPAS2E1, TLB Invalidate by Intermediate Physical Address, Stage 2, EL1

The TLBI IPAS2E1 characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 2 only translation table entry, from any level of the translation table walk.
- The entry would be required to translate the specified IPA using the EL1&0 translation regime.
- The entry would be used with the current VMID.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to the PE that executes this System instruction.

For more information about the architectural requirements for this System instruction, see 'Invalidating TLB entries from stage 2 translations'.

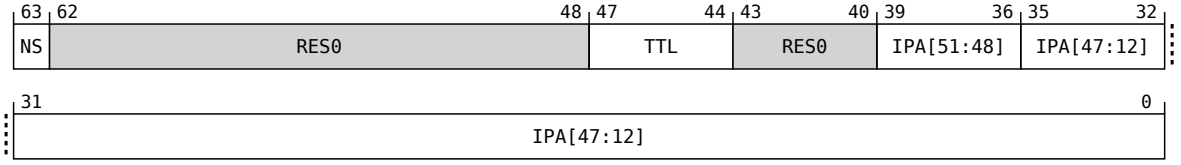
Configuration

There are no configuration notes.

Attributes

TLBI IPAS2E1 is a 64-bit System instruction.

Field descriptions



NS, bit [63]

When FEAT_SEL2 is implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
0b00xx	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
0b01xx	The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.
0b10xx	The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Reserved. Treat as if TTL<3:2> is 0b00. 0b10 : Level 2. 0b11 : Level 3.
0b11xx	The entry comes from a 64KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:40]

Reserved, RES0.

IPA[51:48], bits [39:36]

When FEAT_LPA is implemented:

Extension to IPA[47:12]. For more information, see IPA[47:12].

Otherwise:

Reserved, RES0.

IPA[47:12], bits [35:0]

Bits[47:12] of the intermediate physical address to match. For implementations with fewer than 48 bits, the upper bits of this field are RES0.

If ID_AA64MMFR0_EL1.PARange is 0b0110, bits IPA[51:48] form the upper part of the address value. Otherwise and bits IPA[51:48] are RES0.

Executing TLBI IPAS2E1

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI IPAS2E1{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0100	0b001

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
        ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.14 TLBI IPAS2E1IS, TLB Invalidate by Intermediate Physical Address, Stage 2, EL1, Inner Shareable

The TLBI IPAS2E1IS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 2 only translation table entry, from any level of the translation table walk.
- The entry would be required to translate the specified IPA using the EL1&0 translation regime.
- The entry would be used with the current VMID.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

For more information about the architectural requirements for this System instruction, see ‘Invalidating TLB entries from stage 2 translations’.

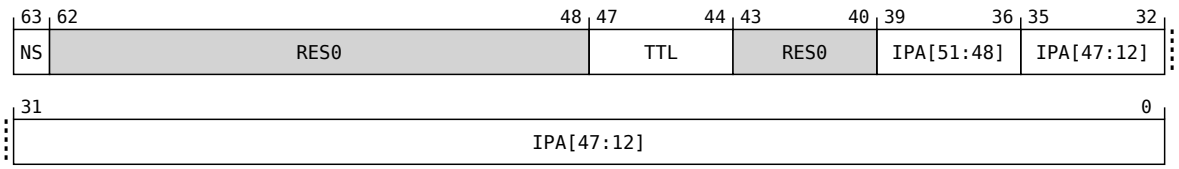
Configuration

There are no configuration notes.

Attributes

TLBI IPAS2E1IS is a 64-bit System instruction.

Field descriptions



NS, bit [63]

When FEAT_SEL2 is implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
0b00xx	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
0b01xx	The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.
0b10xx	The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Reserved. Treat as if TTL<3:2> is 0b00. 0b10 : Level 2. 0b11 : Level 3.
0b11xx	The entry comes from a 64KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:40]

Reserved, RES0.

IPA[51:48], bits [39:36]

When FEAT_LPA is implemented:

Extension to IPA[47:12]. For more information, see IPA[47:12].

Otherwise:

Reserved, RES0.

IPA[47:12], bits [35:0]

Bits[47:12] of the intermediate physical address to match. For implementations with fewer than 48 bits, the upper bits of this field are RES0.

If ID_AA64MMFR0_EL1.PARange is 0b0110, bits IPA[51:48] form the upper part of the address value. Otherwise and bits IPA[51:48] are RES0.

Executing TLBI IPAS2E1IS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI IPAS2E1IS{, <Xt>}
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
        ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.15 TLBI IPAS2E1OS, TLB Invalidate by Intermediate Physical Address, Stage 2, EL1, Outer Shareable

The TLBI IPAS2E1OS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 2 only translation table entry, from any level of the translation table walk.
- The entry would be required to translate the specified IPA using the EL1&0 translation regime.
- The entry would be used with the current VMID.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

For more information about the architectural requirements for this System instruction, see ‘Invalidating TLB entries from stage 2 translations’.

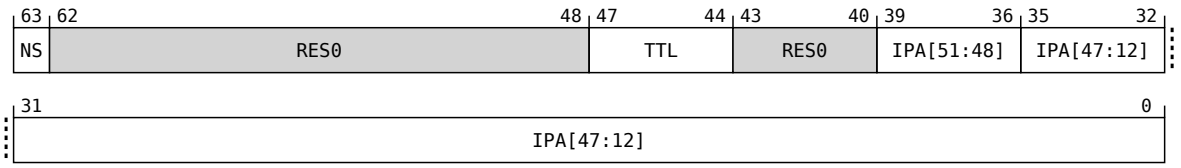
Configuration

This system instruction is present only when FEAT_TLBIOS is implemented. Otherwise, direct accesses to TLBI IPAS2E1OS are UNDEFINED.

Attributes

TLBI IPAS2E1OS is a 64-bit System instruction.

Field descriptions



NS, bit [63]

When FEAT_SEL2 is implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TTL, bits [47:44]

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
0b00xx	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
0b01xx	The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.
0b10xx	The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Reserved. Treat as if TTL<3:2> is 0b00. 0b10 : Level 2. 0b11 : Level 3.
0b11xx	The entry comes from a 64KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:40]

Reserved, RES0.

IPA[51:48], bits [39:36]

Extension to IPA[47:12]. For more information, see IPA[47:12].

IPA[47:12], bits [35:0]

Bits[47:12] of the intermediate physical address to match. For implementations with fewer than 48 bits, the upper bits of this field are RES0.

If ID_AA64MMFR0_EL1.PARange is 0b0110, bits IPA[51:48] form the upper part of the address value. Otherwise and bits IPA[51:48] are RES0.

Executing TLBI IPAS2E1OS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI IPAS2E1OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0100	0b000

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_TLBIOS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
        ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.16 TLBI IPAS2LE1, TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1

The TLBI IPAS2LE1 characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 2 only translation table entry, from the final level of the translation table walk.
- The entry would be required to translate the specified IPA using the EL1&0 translation regime.
- The entry would be used with the current VMID.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to the PE that executes this System instruction.

For more information about the architectural requirements for this System instruction, see 'Invalidating TLB entries from stage 2 translations'.

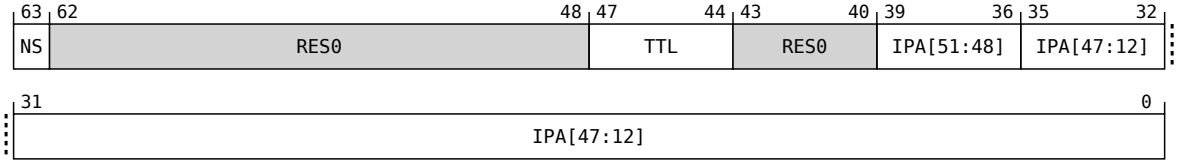
Configuration

There are no configuration notes.

Attributes

TLBI IPAS2LE1 is a 64-bit System instruction.

Field descriptions



NS, bit [63]

When FEAT_SEL2 is implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
0b00xx	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
0b01xx	The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.
0b10xx	The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Reserved. Treat as if TTL<3:2> is 0b00. 0b10 : Level 2. 0b11 : Level 3.
0b11xx	The entry comes from a 64KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:40]

Reserved, RES0.

IPA[51:48], bits [39:36]

When FEAT_LPA is implemented:

Extension to IPA[47:12]. For more information, see IPA[47:12].

Otherwise:

Reserved, RES0.

IPA[47:12], bits [35:0]

Bits[47:12] of the intermediate physical address to match. For implementations with fewer than 48 bits, the upper bits of this field are RES0.

If ID_AA64MMFR0_EL1.PARange is 0b0110, bits IPA[51:48] form the upper part of the address value. Otherwise and bits IPA[51:48] are RES0.

Executing TLBI IPAS2LE1

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI IPAS2LE1{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0100	0b101

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Last,
        ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.17 TLBI IPAS2LE1IS, TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Inner Shareable

The TLBI IPAS2LE1IS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 2 only translation table entry, from the final level of the translation table walk.
- The entry would be required to translate the specified IPA using the EL1&0 translation regime.
- The entry would be used with the current VMID.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

For more information about the architectural requirements for this System instruction, see ‘Invalidating TLB entries from stage 2 translations’.

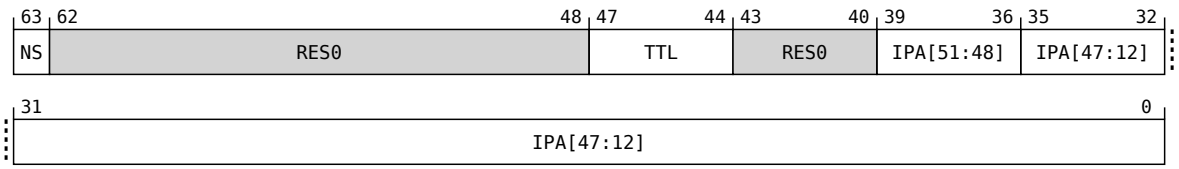
Configuration

There are no configuration notes.

Attributes

TLBI IPAS2LE1IS is a 64-bit System instruction.

Field descriptions



NS, bit [63]

When FEAT_SEL2 is implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
0b00xx	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
0b01xx	The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.
0b10xx	The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Reserved. Treat as if TTL<3:2> is 0b00. 0b10 : Level 2. 0b11 : Level 3.
0b11xx	The entry comes from a 64KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:40]

Reserved, RES0.

IPA[51:48], bits [39:36]

When FEAT_LPA is implemented:

Extension to IPA[47:12]. For more information, see IPA[47:12].

Otherwise:

Reserved, RES0.

IPA[47:12], bits [35:0]

Bits[47:12] of the intermediate physical address to match. For implementations with fewer than 48 bits, the upper bits of this field are RES0.

If ID_AA64MMFR0_EL1.PARange is 0b0110, bits IPA[51:48] form the upper part of the address value. Otherwise and bits IPA[51:48] are RES0.

Executing TLBI IPAS2LE1IS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI IPAS2LE1IS{, <Xt>}
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0000	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Last,
        ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.18 TLBI IPAS2LE1OS, TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Outer Shareable

The TLBI IPAS2LE1OS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 2 only translation table entry, from the final level of the translation table walk.
- The entry would be required to translate the specified IPA using the EL1&0 translation regime.
- The entry would be used with the current VMID.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

For more information about the architectural requirements for this System instruction, see ‘Invalidating TLB entries from stage 2 translations’.

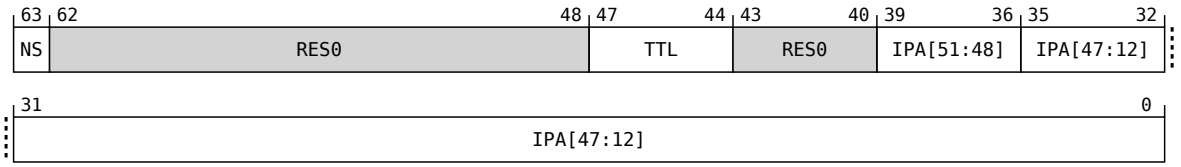
Configuration

This system instruction is present only when FEAT_TLBIOS is implemented. Otherwise, direct accesses to TLBI IPAS2LE1OS are UNDEFINED.

Attributes

TLBI IPAS2LE1OS is a 64-bit System instruction.

Field descriptions



NS, bit [63]

When FEAT_SEL2 is implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TTL, bits [47:44]

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
0b00xx	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
0b01xx	The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.
0b10xx	The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Reserved. Treat as if TTL<3:2> is 0b00. 0b10 : Level 2. 0b11 : Level 3.
0b11xx	The entry comes from a 64KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

Bits [43:40]

Reserved, RES0.

IPA[51:48], bits [39:36]

Extension to IPA[47:12]. For more information, see IPA[47:12].

IPA[47:12], bits [35:0]

Bits[47:12] of the intermediate physical address to match. For implementations with fewer than 48 bits, the upper bits of this field are RES0.

If ID_AA64MMFR0_EL1.PARange is 0b0110, bits IPA[51:48] form the upper part of the address value. Otherwise and bits IPA[51:48] are RES0.

Executing TLBI IPAS2LE1OS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI IPAS2LE1OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0100	0b100

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_TLBIOS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_IPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Last,
        ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.19 TLBI RIPAS2E1, TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1

The TLBI RIPAS2E1 characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 2 only translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
- The entry would be required to translate any IPA in the specified address range using the EL1&0 translation regime.
- The entry would be used with the current VMID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to the PE that executes this System instruction.

For translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 0000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 00000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

For more information about the architectural requirements for this System instruction, see ‘Invalidating TLB entries from stage 2 translations’.

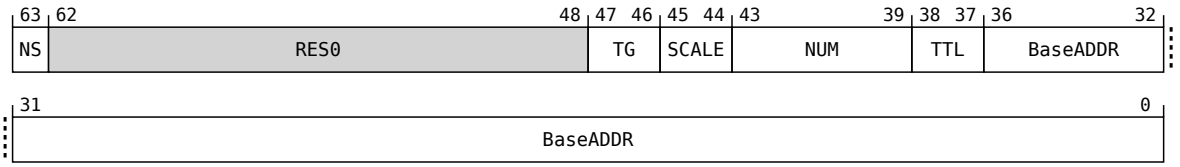
Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented. Otherwise, direct accesses to TLBI RIPAS2E1 are UNDEFINED.

Attributes

TLBI RIPAS2E1 is a 64-bit System instruction.

Field descriptions



NS, bit [63]

When FEAT_SEL2 is implemented:

Not Secure. Specifies the IPA space.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries. When using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Executing TLBI RIPAS2E1

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RIPAS2E1{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0100	0b010

```
if !IsFeatureImplemented(FEAT_TLBIRANGE) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
        ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.20 TLBI RIPAS2E1IS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1, Inner Shareable

The TLBI RIPAS2E1IS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 2 only translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
- The entry would be required to translate any IPA in the specified address range using the EL1&0 translation regime.
- The entry would be used with the current VMID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

For translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

For more information about the architectural requirements for this System instruction, see ‘Invalidating TLB entries from stage 2 translations’.

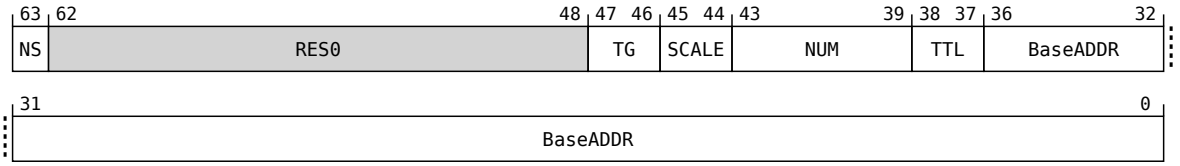
Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented. Otherwise, direct accesses to TLBI RIPAS2E1IS are UNDEFINED.

Attributes

TLBI RIPAS2E1IS is a 64-bit System instruction.

Field descriptions



NS, bit [63]

When FEAT_SEL2 is implemented:

Not Secure. Specifies the IPA space.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries. When using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Executing TLBI RIPAS2E1IS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RIPAS2E1IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0000	0b010

```
if !IsFeatureImplemented(FEAT_TLBIRANGE) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
        ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.21 TLBI RIPAS2E1OS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1, Outer Shareable

The TLBI RIPAS2E1OS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 2 only translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
- The entry would be required to translate any IPA in the specified address range using the EL1&0 translation regime.
- The entry would be used with the current VMID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

For translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

For more information about the architectural requirements for this System instruction, see ‘Invalidating TLB entries from stage 2 translations’.

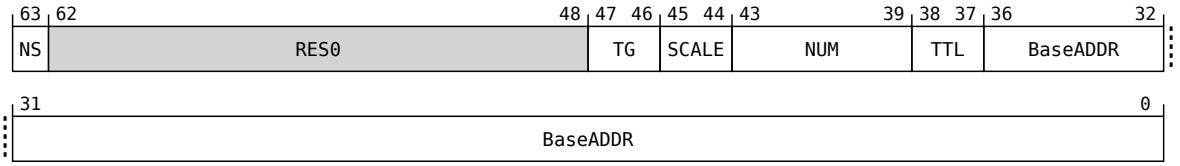
Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_TLBIOS is implemented. Otherwise, direct accesses to TLBI RIPAS2E1OS are UNDEFINED.

Attributes

TLBI RIPAS2E1OS is a 64-bit System instruction.

Field descriptions



NS, bit [63]

When FEAT_SEL2 is implemented:

Not Secure. Specifies the IPA space.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries. When using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Executing TLBI RIPAS2E1OS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RIPAS2E1OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0100	0b011

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_TLBIOS)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
        ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.22 TLBI RIPAS2LE1, TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1

The TLBI RIPAS2LE1 characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 2 only translation table entry, from the leaf level of the translation table walk, indicated by the TTL hint.
- The entry would be required to translate any IPA in the specified address range using the EL1&0 translation regime.
- The entry would be used with the current VMID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation only applies to the PE that executes this System instruction.

For translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 0000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 00000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

For more information about the architectural requirements for this System instruction, see ‘Invalidating TLB entries from stage 2 translations’.

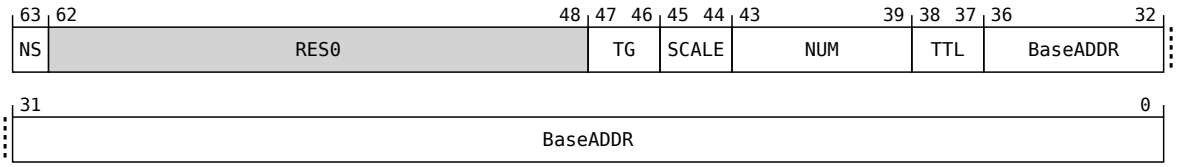
Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented. Otherwise, direct accesses to TLBI RIPAS2LE1 are UNDEFINED.

Attributes

TLBI RIPAS2LE1 is a 64-bit System instruction.

Field descriptions



NS, bit [63]

When FEAT_SEL2 is implemented:

Not Secure. Specifies the IPA space.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries. When using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Executing TLBI RIPAS2LE1

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`TLBI RIPAS2LE1{, <Xt>}`

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0100	0b110

```
if !IsFeatureImplemented(FEAT_TLBIRANGE) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Last,
        ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.23 TLBI RIPAS2LE1IS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Inner Shareable

The TLBI RIPAS2LE1IS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 2 only translation table entry, from the leaf level of the translation table walk, indicated by the TTL hint.
- The entry would be required to translate any IPA in the specified address range using the EL1&0 translation regime.
- The entry would be used with the current VMID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

For translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

For more information about the architectural requirements for this System instruction, see ‘Invalidating TLB entries from stage 2 translations’.

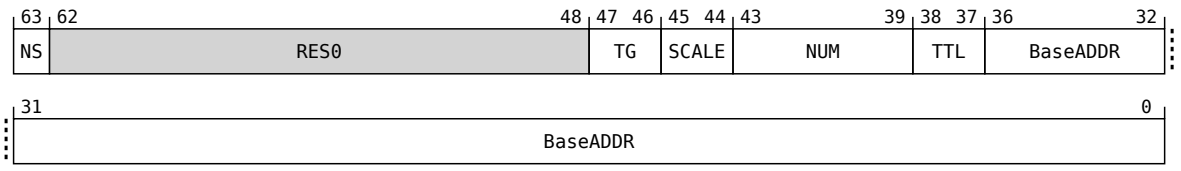
Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented. Otherwise, direct accesses to TLBI RIPAS2LE1IS are UNDEFINED.

Attributes

TLBI RIPAS2LE1IS is a 64-bit System instruction.

Field descriptions



NS, bit [63]

When FEAT_SEL2 is implemented:

Not Secure. Specifies the IPA space.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries. When using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Executing TLBI RIPAS2LE1IS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`TLBI RIPAS2LE1IS{, <Xt>}`

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0000	0b110

```
if !IsFeatureImplemented(FEAT_TLBIRANGE) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Last,
        ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.24 TLBI RIPAS2LE1OS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Outer Shareable

The TLBI RIPAS2LE1OS characteristics are:

Purpose

If EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 2 only translation table entry, from the leaf level of the translation table walk, indicated by the TTL hint.
- The entry would be required to translate any IPA in the specified address range using the EL1&0 translation regime.
- The entry would be used with the current VMID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, then:

- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

The invalidation is not required to apply to caching structures that combine stage 1 and stage 2 translation table entries.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

For translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

For more information about the architectural requirements for this System instruction, see ‘Invalidating TLB entries from stage 2 translations’.

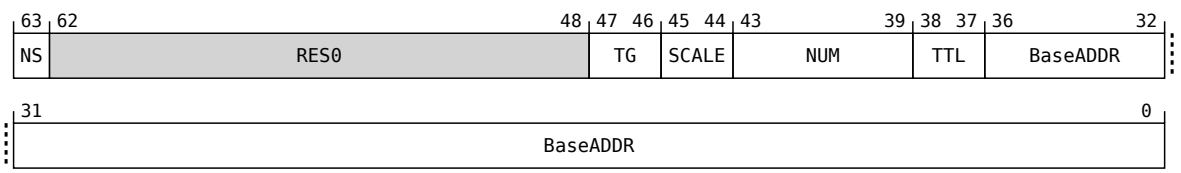
Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_TLBIOS is implemented. Otherwise, direct accesses to TLBI RIPAS2LE1OS are UNDEFINED.

Attributes

TLBI RIPAS2LE1OS is a 64-bit System instruction.

Field descriptions



Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NS, bit [63]

When FEAT_SEL2 is implemented:

Not Secure. Specifies the IPA space.

NS	Meaning
0b0	IPA is in the Secure IPA space.
0b1	IPA is in the Non-secure IPA space.

Otherwise:

Reserved, RES0.

Bits [62:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries. When using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RIPAS2LE1OS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`TLBI RIPAS2LE1OS{, <Xt>}`

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0100	0b111

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_TLBIOS)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_RIPAS2(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Last,
        ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.25 TLBI RVAAE1, TLB Range Invalidate by VA, All ASID, EL1

The TLBI RVAAE1 characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The invalidation applies to the PE that executes this System instruction.

Note

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

For translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

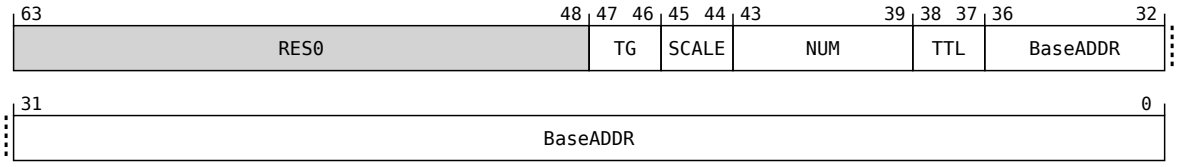
Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented. Otherwise, direct accesses to TLBI RVAAE1 are UNDEFINED.

Attributes

TLBI RVAAE1 is a 64-bit System instruction.

Field descriptions



Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries. When using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVAAE1

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVAAE1{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0110	0b011

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_TLBIRANGE) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HCR_EL2.FB == '1' then
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
            ↪TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
        ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.26 TLBI RVAAE1IS, TLB Range Invalidate by VA, All ASID, EL1, Inner Shareable

The TLBI RVAAE1IS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, then:

- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

For translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 00000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

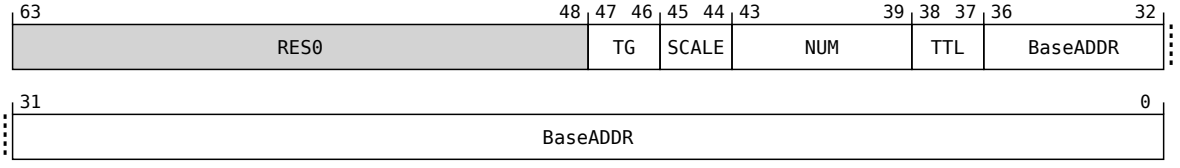
Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented. Otherwise, direct accesses to TLBI RVAAE1IS are UNDEFINED.

Attributes

TLBI RVAAE1IS is a 64-bit System instruction.

Field descriptions



Bits [63:48]

Reserved, RES0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries. When using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVAAE1IS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVAAE1IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0010	0b011

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_TLBIRANGE) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
        ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.27 TLBI RVAAE1OS, TLB Range Invalidate by VA, All ASID, EL1, Outer Shareable

The TLBI RVAAE1OS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, then:

- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

For translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 00000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

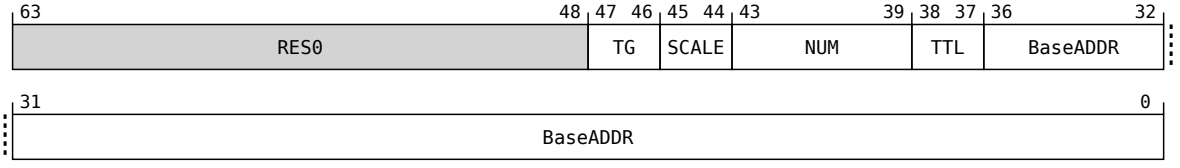
Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_TLBIOS is implemented. Otherwise, direct accesses to TLBI RVAAE1OS are UNDEFINED.

Attributes

TLBI RVAAE1OS is a 64-bit System instruction.

Field descriptions



Bits [63:48]

Reserved, RES0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries. When using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVAAE1OS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVAAE1OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0101	0b011

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_TLBIOS)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
        ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.28 TLBI RVAALE1, TLB Range Invalidate by VA, All ASID, Last level, EL1

The TLBI RVAALE1 characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from the leaf level of the translation table walk, indicated by the TTL hint.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The invalidation applies to the PE that executes this System instruction.

Note

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

For translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 0000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 00000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

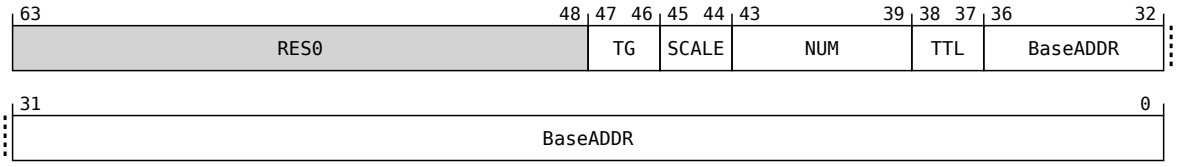
Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented. Otherwise, direct accesses to TLBI RVAALE1 are UNDEFINED.

Attributes

TLBI RVAALE1 is a 64-bit System instruction.

Field descriptions



Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries. When using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVAALE1

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVAALE1{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0110	0b111

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_TLBIRANGE) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HCR_EL2.FB == '1' then
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
            ↪TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Last,
        ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.29 TLBI RVAALE1IS, TLB Range Invalidate by VA, All ASID, Last Level, EL1, Inner Shareable

The TLBI RVAALE1IS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from the leaf level of the translation table walk, indicated by the TTL hint.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, then:

- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

For translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 00000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

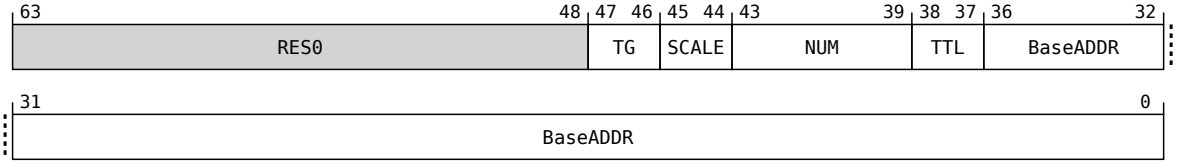
Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented. Otherwise, direct accesses to TLBI RVAALE1IS are UNDEFINED.

Attributes

TLBI RVAALE1IS is a 64-bit System instruction.

Field descriptions



Bits [63:48]

Reserved, RES0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries. When using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVAALE1IS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVAALE1IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0010	0b111

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_TLBIRANGE) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Last,
        ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.30 TLBI RVAALE1OS, TLB Range Invalidate by VA, All ASID, Last Level, EL1, Outer Shareable

The TLBI RVAALE1OS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from the leaf level of the translation table walk, indicated by the TTL hint.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, then:

- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

For translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 00000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

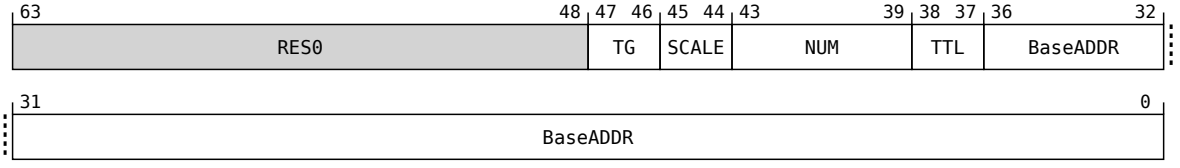
Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_TLBIOS is implemented. Otherwise, direct accesses to TLBI RVAALE1OS are UNDEFINED.

Attributes

TLBI RVAALE1OS is a 64-bit System instruction.

Field descriptions



Bits [63:48]

Reserved, RES0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries. When using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVAALE1OS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVAALE1OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0101	0b111

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_TLBIOS)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_RVAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Last,
        ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.31 TLBI RVAE1, TLB Range Invalidate by VA, EL1

The TLBI RVAE1 characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
- The entry would be used to translate any of the VAs in the specified address range, and one of the following applies:
  - The entry is from a level of lookup above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The invalidation applies to the PE that executes this System instruction.

For translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

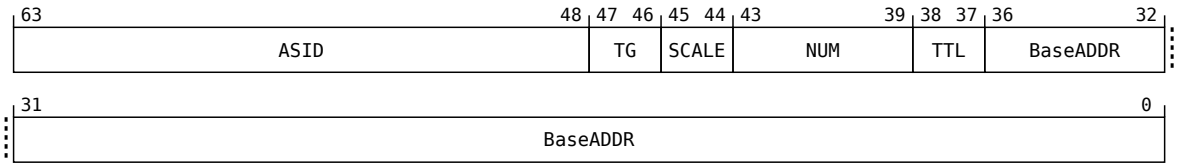
Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented. Otherwise, direct accesses to TLBI RVAE1 are UNDEFINED.

Attributes

TLBI RVAE1 is a 64-bit System instruction.

Field descriptions



ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries. When using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVAE1

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVAE1{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0110	0b001

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_TLBIRANGE) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HCR_EL2.FB == '1' then
        AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
            ↪TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
        ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.32 TLBI RVAE1IS, TLB Range Invalidate by VA, EL1, Inner Shareable

The TLBI RVAE1IS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
- The entry would be used to translate any of the VAs in the specified address range, and one of the following applies:
  - The entry is from a level of lookup above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, then:

- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

For translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

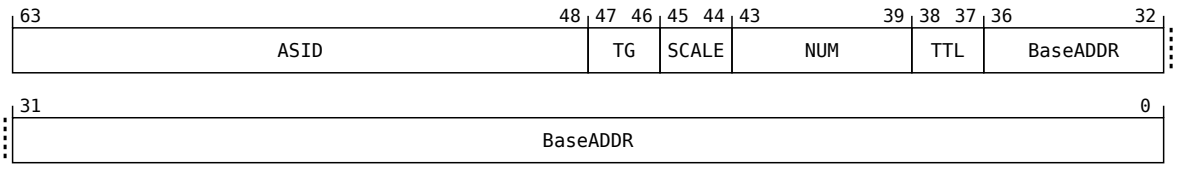
Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented. Otherwise, direct accesses to TLBI RVAE1IS are UNDEFINED.

Attributes

TLBI RVAE1IS is a 64-bit System instruction.

Field descriptions



Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries. When using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Executing TLBI RVAE1IS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`TLBI RVAE1IS{, <Xt>}`

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0010	0b001

```
if !IsFeatureImplemented(FEAT_TLBIRANGE) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
         ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.33 TLBI RVAE1OS, TLB Range Invalidate by VA, EL1, Outer Shareable

The TLBI RVAE1OS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
- The entry would be used to translate any of the VAs in the specified address range, and one of the following applies:
  - The entry is from a level of lookup above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, then:

- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

For translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

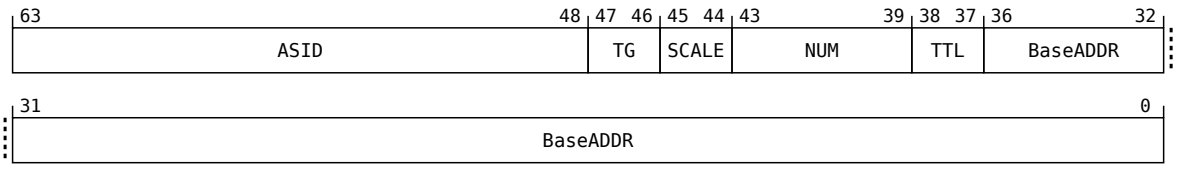
Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_TLBIOS is implemented. Otherwise, direct accesses to TLBI RVAE1OS are UNDEFINED.

Attributes

TLBI RVAE1OS is a 64-bit System instruction.

Field descriptions



Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries. When using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Executing TLBI RVAE1OS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`TLBI RVAE1OS{, <Xt>}`

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0101	0b001

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_TLBIOS)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
         ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.34 TLBI RVAE2, TLB Range Invalidate by VA, EL2

The TLBI RVAE2 characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
- The entry would be used to translate any VA in the range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)] using the EL2 translation regime for the Security state.
- The entry is from any level of the translation table walk.

The invalidation applies to the PE that executes this System instruction.

For translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 0000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 00000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

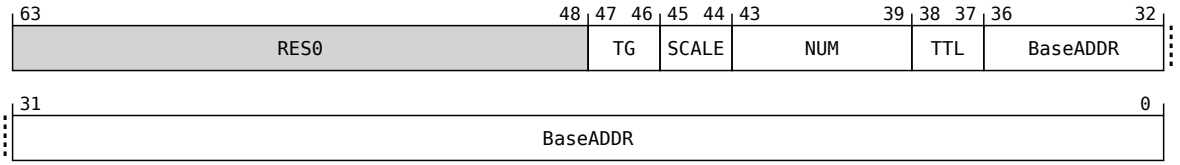
Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented. Otherwise, direct accesses to TLBI RVAE2 are UNDEFINED.

Attributes

TLBI RVAE2 is a 64-bit System instruction.

Field descriptions



Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TG	Meaning
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries. When using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVAE2

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`TLBI RVAE2{, <Xt>}`

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0110	0b001

```
if !IsFeatureImplemented(FEAT_TLBIRANGE) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
        ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.35 TLBI RVAE2IS, TLB Range Invalidate by VA, EL2, Inner Shareable

The TLBI RVAE2IS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
- The entry would be used to translate any VA in the range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)] using the EL2 translation regime for the Security state.
- The entry is from any level of the translation table walk.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

For translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

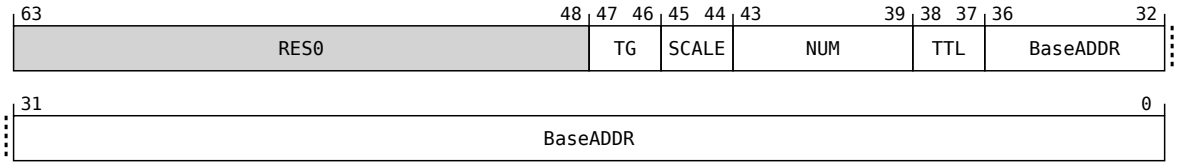
Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented. Otherwise, direct accesses to TLBI RVAE2IS are UNDEFINED.

Attributes

TLBI RVAE2IS is a 64-bit System instruction.

Field descriptions



Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TG	Meaning
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries. When using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVAE2IS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVAE2IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0010	0b001

```
if !IsFeatureImplemented(FEAT_TLBIRANGE) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
        ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.36 TLBI RVAE2OS, TLB Range Invalidate by VA, EL2, Outer Shareable

The TLBI RVAE2OS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
- The entry would be used to translate any VA in the range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)] using the EL2 translation regime for the Security state.
- The entry is from any level of the translation table walk.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

For translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

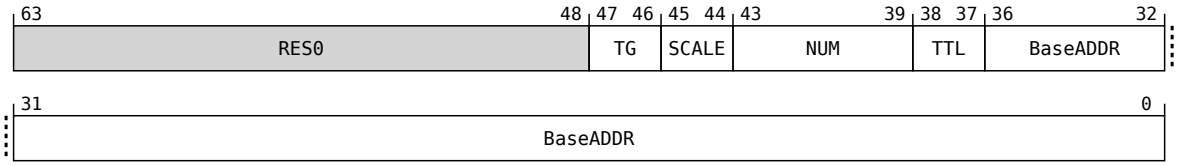
Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_TLBIOS is implemented. Otherwise, direct accesses to TLBI RVAE2OS are UNDEFINED.

Attributes

TLBI RVAE2OS is a 64-bit System instruction.

Field descriptions



Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TG	Meaning
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries. When using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVAE2OS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVAE2OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0101	0b001

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_TLBIOS)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH, TLBILevel_Any,
        ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.37 TLBI RVAE3, TLB Range Invalidate by VA, EL3

The TLBI RVAE3 characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
- The entry would be used to translate any of the VAs in the specified address range using the EL3 translation regime.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation applies to the PE that executes this System instruction.

For translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 00000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

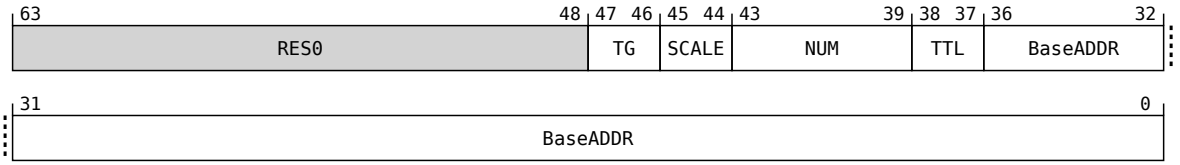
Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented. Otherwise, direct accesses to TLBI RVAE3 are UNDEFINED.

Attributes

TLBI RVAE3 is a 64-bit System instruction.

Field descriptions



Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TG	Meaning
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries. When using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVAE3

This system instruction is an alias of the SYS instruction.

This system instruction encoding is Undefined()

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVAE3{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0110	0b001

```
if !IsFeatureImplemented(FEAT_TLBIRANGE) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.38 TLBI RVAE3IS, TLB Range Invalidate by VA, EL3, Inner Shareable

The TLBI RVAE3IS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
- The entry would be used to translate any of the VAs in the specified address range using the EL3 translation regime.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

For translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

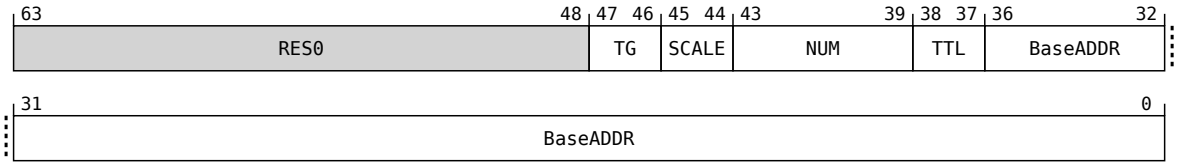
Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented. Otherwise, direct accesses to TLBI RVAE3IS are UNDEFINED.

Attributes

TLBI RVAE3IS is a 64-bit System instruction.

Field descriptions



Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TG	Meaning
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries. When using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVAE3IS

This system instruction is an alias of the SYS instruction.

This system instruction encoding is Undefined()

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVAE3IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0010	0b001

```
if !IsFeatureImplemented(FEAT_TLBIRANGE) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.39 TLBI RVAE3OS, TLB Range Invalidate by VA, EL3, Outer Shareable

The TLBI RVAE3OS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
- The entry would be used to translate any of the VAs in the specified address range using the EL3 translation regime.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

For translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

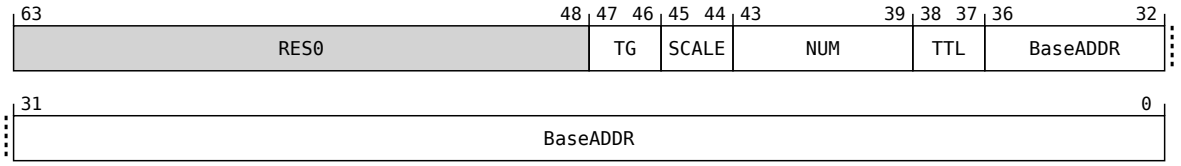
Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_TLBIOS is implemented. Otherwise, direct accesses to TLBI RVAE3OS are UNDEFINED.

Attributes

TLBI RVAE3OS is a 64-bit System instruction.

Field descriptions



Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TG	Meaning
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries. When using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVAE3OS

This system instruction is an alias of the SYS instruction.

This system instruction encoding is Undefined()

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVAE3OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0101	0b001

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_TLBIOS)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.40 TLBI RVALE1, TLB Range Invalidate by VA, Last level, EL1

The TLBI RVALE1 characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
- The entry would be used to translate any of the VAs in the specified address range, and one of the following applies:
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The invalidation applies to the PE that executes this System instruction.

For translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

For more information about the architectural requirements for this System instruction, see ‘Invalidating TLB entries from stage 2 translations’.

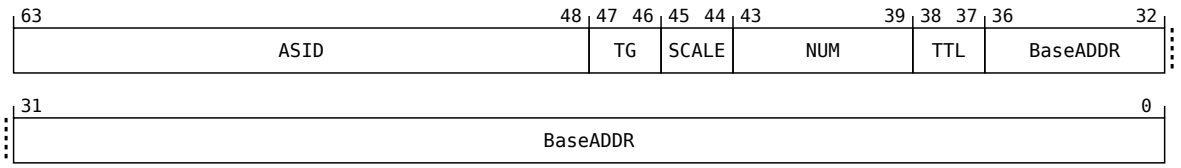
Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented. Otherwise, direct accesses to TLBI RVALE1 are UNDEFINED.

Attributes

TLBI RVALE1 is a 64-bit System instruction.

Field descriptions



ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries. When using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVALE1

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVALE1{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0110	0b101

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_TLBIRANGE) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HCR_EL2.FB == '1' then
        AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
            ↪TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Last,
        ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.41 TLBI RVALE1IS, TLB Range Invalidate by VA, Last level, EL1, Inner Shareable

The TLBI RVALE1IS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
- The entry would be used to translate any of the VAs in the specified address range, and one of the following applies:
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, then:

- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

For translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

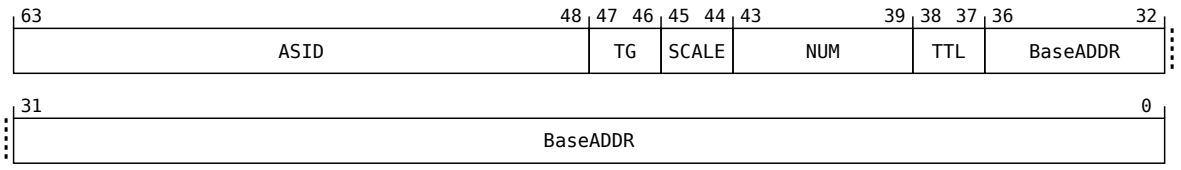
Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented. Otherwise, direct accesses to TLBI RVALE1IS are UNDEFINED.

Attributes

TLBI RVALE1IS is a 64-bit System instruction.

Field descriptions



ASID, bits [63:48]

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries. When using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Executing TLBI RVALE1IS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVALE1IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0010	0b101

```
if !IsFeatureImplemented(FEAT_TLBIRANGE) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Last,
         ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.42 TLBI RVALE1OS, TLB Range Invalidate by VA, Last level, EL1, Outer Shareable

The TLBI RVALE1OS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
- The entry would be used to translate any of the VAs in the specified address range, and one of the following applies:
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The entry would be used with the current VMID and would be required to translate any of the VAs in the specified address range using the EL1&0 translation regime for the Security state.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, then:

- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

For translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

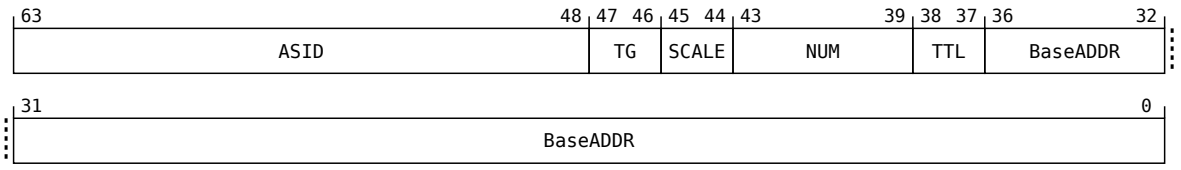
Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_TLBIOS is implemented. Otherwise, direct accesses to TLBI RVALE1OS are UNDEFINED.

Attributes

TLBI RVALE1OS is a 64-bit System instruction.

Field descriptions



ASID, bits [63:48]

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries. When using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Executing TLBI RVALE1OS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`TLBI RVALE1OS{, <Xt>}`

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0101	0b101

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_TLBIOS)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_RVA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Last,
         ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.43 TLBI RVALE2, TLB Range Invalidate by VA, Last level, EL2

The TLBI RVALE2 characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
- The entry would be used to translate any VA in the range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)] using the EL2 translation regime for the Security state.
- The entry is from the final level of the translation table walk.

The invalidation applies to the PE that executes this System instruction.

For translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 0000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 00000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

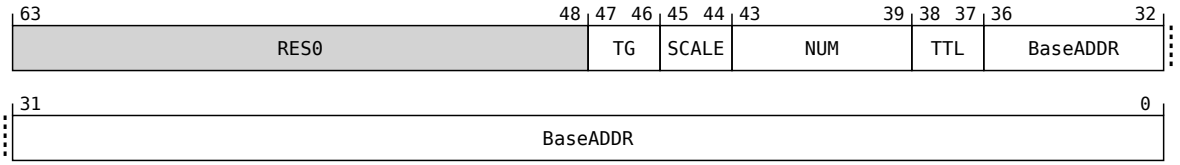
Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented. Otherwise, direct accesses to TLBI RVALE2 are UNDEFINED.

Attributes

TLBI RVALE2 is a 64-bit System instruction.

Field descriptions



Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TG	Meaning
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries. When using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVALE2

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVALE2{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0110	0b101

```
if !IsFeatureImplemented(FEAT_TLBIRANGE) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH, TLBILevel_Last,
        ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.44 TLBI RVALE2IS, TLB Range Invalidate by VA, Last level, EL2, Inner Shareable

The TLBI RVALE2IS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
- The entry would be used to translate any VA in the range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)] using the EL2 translation regime for the Security state.
- The entry is from the final level of the translation table walk.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

For translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

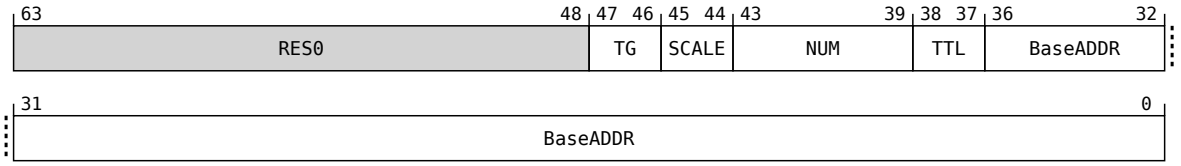
Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented. Otherwise, direct accesses to TLBI RVALE2IS are UNDEFINED.

Attributes

TLBI RVALE2IS is a 64-bit System instruction.

Field descriptions



Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TG	Meaning
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries. When using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVALE2IS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVALE2IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0010	0b101

```
if !IsFeatureImplemented(FEAT_TLBIRANGE) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH, TLBILevel_Last,
        ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.45 TLBI RVALE2OS, TLB Range Invalidate by VA, Last level, EL2, Outer Shareable

The TLBI RVALE2OS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.
- The entry would be used to translate any VA in the range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)] using the EL2 translation regime for the Security state.
- The entry is from the final level of the translation table walk.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

For translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

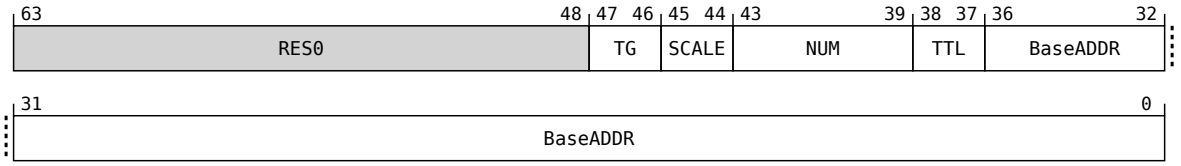
Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_TLBIOS is implemented. Otherwise, direct accesses to TLBI RVALE2OS are UNDEFINED.

Attributes

TLBI RVALE2OS is a 64-bit System instruction.

Field descriptions



Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TG	Meaning
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries. When using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVALE2OS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVALE2OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0101	0b101

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_TLBIOS)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_RVA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH, TLBILevel_Last,
        ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.46 TLBI RVALE3, TLB Range Invalidate by VA, Last level, EL3

The TLBI RVALE3 characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from the final level of the translation table walk up to the level indicated in the TTL hint.
- The entry would be used to translate any of the VAs in the specified address range using the EL3 translation regime.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation applies to the PE that executes this System instruction.

For translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 00000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 00000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

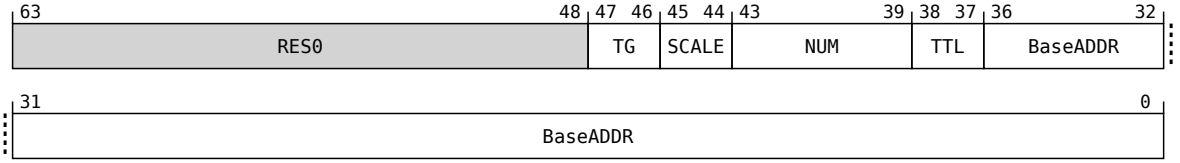
Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented. Otherwise, direct accesses to TLBI RVALE3 are UNDEFINED.

Attributes

TLBI RVALE3 is a 64-bit System instruction.

Field descriptions



Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TG	Meaning
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries. When using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVALE3

This system instruction is an alias of the SYS instruction.

This system instruction encoding is Undefined()

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVALE3{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0110	0b101

```
if !IsFeatureImplemented(FEAT_TLBIRANGE) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.47 TLBI RVALE3IS, TLB Range Invalidate by VA, Last level, EL3, Inner Shareable

The TLBI RVALE3IS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from the final level of the translation table walk up to the level indicated in the TTL hint.
- The entry would be used to translate any of the VAs in the specified address range using the EL3 translation regime.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

For translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

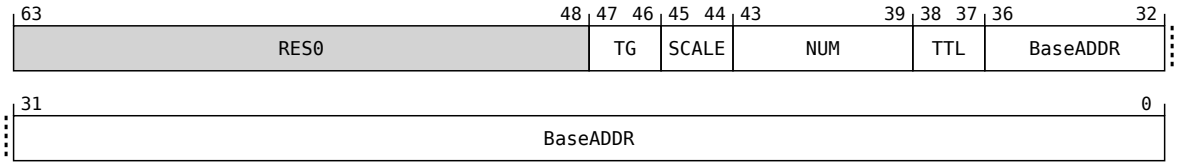
Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented. Otherwise, direct accesses to TLBI RVALE3IS are UNDEFINED.

Attributes

TLBI RVALE3IS is a 64-bit System instruction.

Field descriptions



Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TG	Meaning
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries. When using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVALE3IS

This system instruction is an alias of the SYS instruction.

This system instruction encoding is Undefined()

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVALE3IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0010	0b101

```
if !IsFeatureImplemented(FEAT_TLBIRANGE) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.48 TLBI RVALE3OS, TLB Range Invalidate by VA, Last level, EL3, Outer Shareable

The TLBI RVALE3OS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from the final level of the translation table walk up to the level indicated in the TTL hint.
- The entry would be used to translate any of the VAs in the specified address range using the EL3 translation regime.
- The entry is within the address range determined by the formula [BaseADDR <= VA < BaseADDR + ((NUM +1)*$2^{(5*SCALE+1)}$ * Translation_Granule_Size)].

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

For translation table entry, the range of addresses invalidated is UNPREDICTABLE when:

- For the 4K translation granule:
  - If TTL==01 and BaseADDR[29:12] is not equal to 000000000000000000.
  - If TTL==10 and BaseADDR[20:12] is not equal to 000000000.
- For the 16K translation granule:
  - If TTL==10 and BaseADDR[24:14] is not equal to 00000000000.
- For the 64K translation granule:
  - If TTL==01 and BaseADDR[41:16] is not equal to 000000000000000000000000000000.
  - If TTL==10 and BaseADDR[28:16] is not equal to 0000000000000.

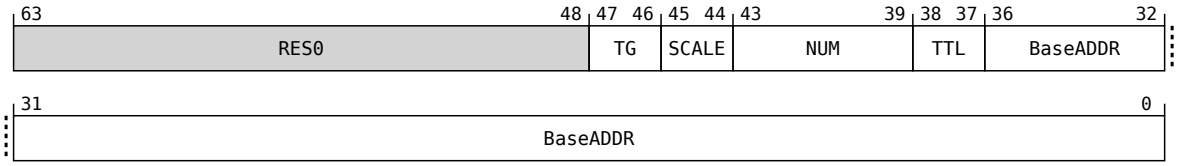
Configuration

This system instruction is present only when FEAT_TLBIRANGE is implemented and FEAT_TLBIOS is implemented. Otherwise, direct accesses to TLBI RVALE3OS are UNDEFINED.

Attributes

TLBI RVALE3OS is a 64-bit System instruction.

Field descriptions



Bits [63:48]

Reserved, RES0.

TG, bits [47:46]

Translation granule size.

TG	Meaning
0b00	Reserved.
0b01	4K translation granule.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TG	Meaning
0b10	16K translation granule.
0b11	64K translation granule.

The instruction takes a translation granule size for the translations that are being invalidated. If the translations used a different translation granule size than the one being specified, then the architecture does not require that the instruction invalidates any entries.

SCALE, bits [45:44]

The exponent element of the calculation that is used to produce the upper range.

NUM, bits [43:39]

The base element of the calculation that is used to produce the upper range.

TTL, bits [38:37]

TTL Level hint. The TTL hint is only guaranteed to invalidate:

- Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
- Leaf-level entries in the range that match the level described by the TTL hint.

TTL	Meaning
0b00	The entries in the range can be using any level for the translation table entries.
0b01	When using a 4KB or 64KB translation granule, all entries to invalidate are Level 1 translation table entries. When using a 16KB translation granule, this value is reserved and hardware should treat this field as 0b00.
0b10	The TTL hint indicates level 2.
0b11	The TTL hint indicates level 3.

BaseADDR, bits [36:0]

The starting address for the range of the maintenance instruction.

When using a 4KB translation granule, this field is BaseADDR[48:12].

When using a 16KB translation granule, this field is BaseADDR[50:14].

When using a 64KB translation granule, this field is BaseADDR[52:16].

Executing TLBI RVALE3OS

This system instruction is an alias of the SYS instruction.

This system instruction encoding is Undefined()

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI RVALE3OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0101	0b101

```
if !(IsFeatureImplemented(FEAT_TLBIRANGE) && IsFeatureImplemented(FEAT_TLBIOS)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.49 TLBI VAAE1, TLB Invalidate by VA, All ASID, EL1

The TLBI VAAE1 characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.

The entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The invalidation applies to the PE that executes this System instruction.

Note

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

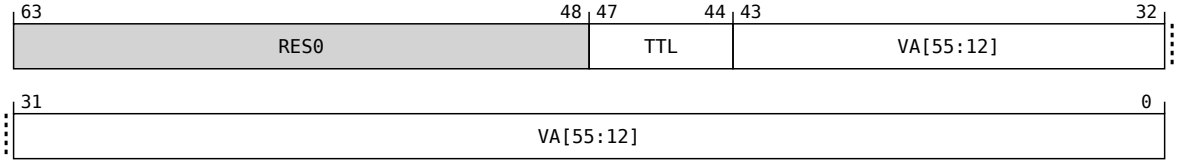
Configuration

There are no configuration notes.

Attributes

TLBI VAAE1 is a 64-bit System instruction.

Field descriptions



Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
0b00xx	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
0b01xx	The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
0b10xx	The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Reserved. Treat as if TTL<3:2> is 0b00. 0b10 : Level 2. 0b11 : Level 3.
0b11xx	The entry comes from a 64KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VAAE1

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`TLBI VAAE1{, <Xt>}`

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0111	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HCR_EL2.FB == '1' then
        AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
            ↪TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
        ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.50 TLBI VAAE1IS, TLB Invalidate by VA, All ASID, EL1, Inner Shareable

The TLBI VAAE1IS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.

The entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

From Armv8.4, when a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, then:

- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

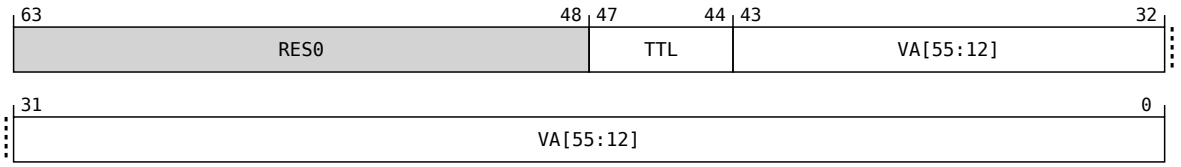
Configuration

There are no configuration notes.

Attributes

TLBI VAAE1IS is a 64-bit System instruction.

Field descriptions



Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
0b00xx	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
0b01xx	The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.
0b10xx	The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Reserved. Treat as if TTL<3:2> is 0b00. 0b10 : Level 2. 0b11 : Level 3.
0b11xx	The entry comes from a 64KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VAAE1IS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`TLBI VAAE1IS{, <Xt>}`

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0011	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
        ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.51 TLBI VAAE1OS, TLB Invalidate by VA, All ASID, EL1, Outer Shareable

The TLBI VAAE1OS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.

The entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, then:

- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

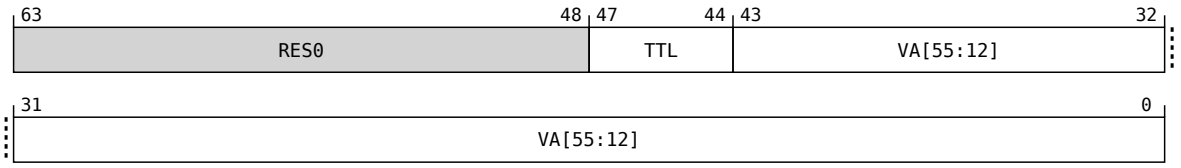
Configuration

This system instruction is present only when FEAT_TLBIOS is implemented. Otherwise, direct accesses to TLBI VAAE1OS are UNDEFINED.

Attributes

TLBI VAAE1OS is a 64-bit System instruction.

Field descriptions



Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
0b00xx	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
0b01xx	The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.
0b10xx	The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Reserved. Treat as if TTL<3:2> is 0b00. 0b10 : Level 2. 0b11 : Level 3.
0b11xx	The entry comes from a 64KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VAAE1OS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`TLBI VAAE1OS{, <Xt>}`

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0001	0b011

```
if !IsFeatureImplemented(FEAT_TLBIOS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
        ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.52 TLBI VAALE1, TLB Invalidate by VA, All ASID, Last level, EL1

The TLBI VAALE1 characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from the final level of the translation table walk.

The entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The invalidation applies to the PE that executes this System instruction.

Note

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

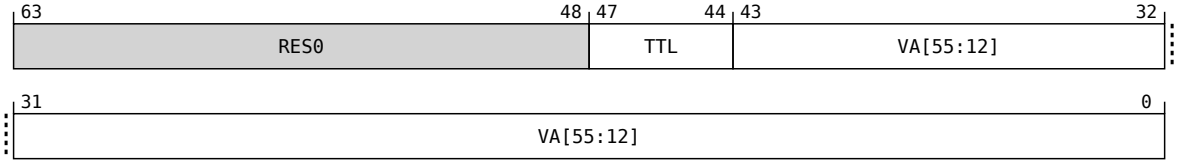
Configuration

There are no configuration notes.

Attributes

TLBI VAALE1 is a 64-bit System instruction.

Field descriptions



Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
0b00xx	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
0b01xx	The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
0b10xx	The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Reserved. Treat as if TTL<3:2> is 0b00. 0b10 : Level 2. 0b11 : Level 3.
0b11xx	The entry comes from a 64KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VAALE1

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`TLBI VAALE1{, <Xt>}`

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0111	0b111

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HCR_EL2.FB == '1' then
        AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
            ↪TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Last,
        ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.53 TLBI VAALE1IS, TLB Invalidate by VA, All ASID, Last Level, EL1, Inner Shareable

The TLBI VAALE1IS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from the final level of the translation table walk.

The entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

From Armv8.4, when a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, then:

- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

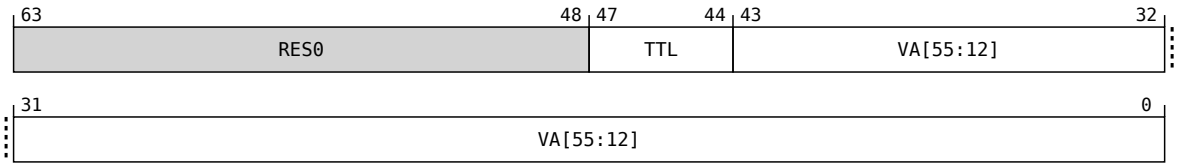
Configuration

There are no configuration notes.

Attributes

TLBI VAALE1IS is a 64-bit System instruction.

Field descriptions



Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
0b00xx	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
0b01xx	The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.
0b10xx	The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Reserved. Treat as if TTL<3:2> is 0b00. 0b10 : Level 2. 0b11 : Level 3.
0b11xx	The entry comes from a 64KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VAALE1IS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`TLBI VAALE1IS{, <Xt>}`

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0011	0b111

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Last,
        ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.54 TLBI VAALE1OS, TLB Invalidate by VA, All ASID, Last Level, EL1, Outer Shareable

The TLBI VAALE1OS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from the final level of the translation table walk.

The entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, then:

- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

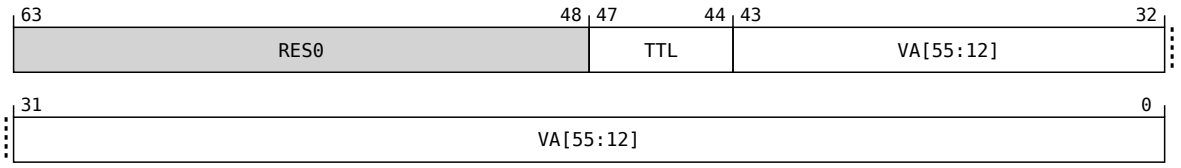
Configuration

This system instruction is present only when FEAT_TLBIOS is implemented. Otherwise, direct accesses to TLBI VAALE1OS are UNDEFINED.

Attributes

TLBI VAALE1OS is a 64-bit System instruction.

Field descriptions



Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
0b00xx	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
0b01xx	The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.
0b10xx	The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Reserved. Treat as if TTL<3:2> is 0b00. 0b10 : Level 2. 0b11 : Level 3.
0b11xx	The entry comes from a 64KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the VA will be affected by this System instruction, regardless of the ASID.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VAALE1OS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`TLBI VAALE1OS{, <Xt>}`

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0001	0b111

```
if !IsFeatureImplemented(FEAT_TLBIOS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_VAA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Last,
        ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.55 TLBI VAE1, TLB Invalidate by VA, EL1

The TLBI VAE1 characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified VA, and one of the following applies:
  - The entry is from a level of lookup above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.

The entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The invalidation applies to the PE that executes this System instruction.

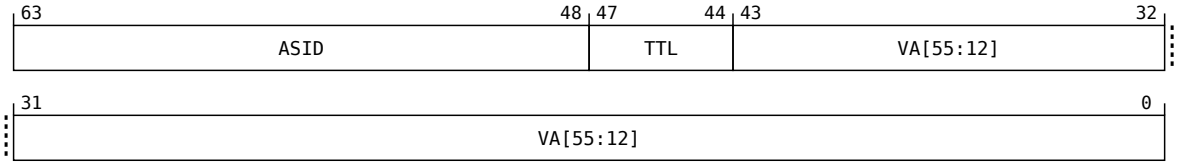
Configuration

There are no configuration notes.

Attributes

TLBI VAE1 is a 64-bit System instruction.

Field descriptions



ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
0b00xx	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
0b01xx	The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.
0b10xx	The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Reserved. Treat as if TTL<3:2> is 0b00. 0b10 : Level 2. 0b11 : Level 3.
0b11xx	The entry comes from a 64KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VAE1

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`TLBI VAE1{, <Xt>}`

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0111	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HCR_EL2.FB == '1' then
        AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, 64]);
    else
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Any,
        ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.56 TLBI VAE1IS, TLB Invalidate by VA, EL1, Inner Shareable

The TLBI VAE1IS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified VA, and one of the following applies:
  - The entry is from a level of lookup above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.

The entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

From Armv8.4, when a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, then:

- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

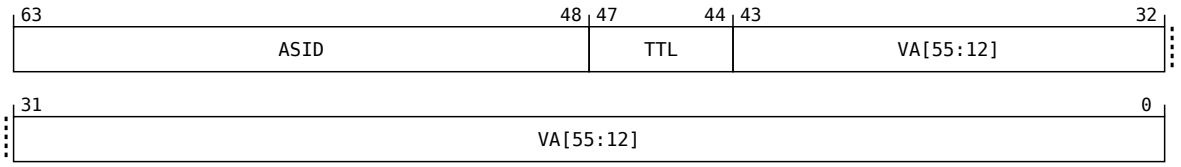
Configuration

There are no configuration notes.

Attributes

TLBI VAE1IS is a 64-bit System instruction.

Field descriptions



ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
0b00xx	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
0b01xx	The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.
0b10xx	The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Reserved. Treat as if TTL<3:2> is 0b00. 0b10 : Level 2. 0b11 : Level 3.
0b11xx	The entry comes from a 64KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VAE1IS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`TLBI VAE1IS{, <Xt>}`

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0011	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Any,
        ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.57 TLBI VAE1OS, TLB Invalidate by VA, EL1, Outer Shareable

The TLBI VAE1OS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified VA, and one of the following applies:
  - The entry is from a level of lookup above the final level and matches the specified ASID.
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.

The entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, then:

- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

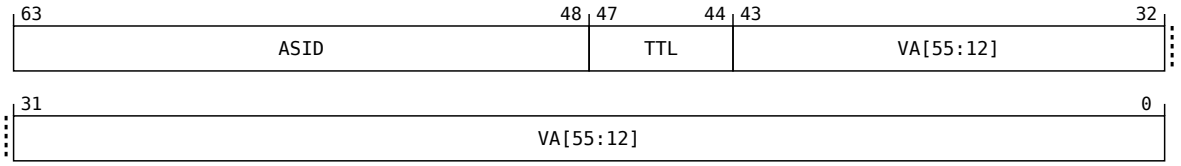
Configuration

This system instruction is present only when FEAT_TLBIOS is implemented. Otherwise, direct accesses to TLBI VAE1OS are UNDEFINED.

Attributes

TLBI VAE1OS is a 64-bit System instruction.

Field descriptions



ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
0b00xx	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
0b01xx	The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.
0b10xx	The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Reserved. Treat as if TTL<3:2> is 0b00. 0b10 : Level 2. 0b11 : Level 3.
0b11xx	The entry comes from a 64KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VAE1OS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`TLBI VAE1OS{, <Xt>}`

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0001	0b001

```
if !IsFeatureImplemented(FEAT_TLBIOS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
            ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Any,
        ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.58 TLBI VAE2, TLB Invalidate by VA, EL2

The TLBI VAE2 characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be required to translate the specified VA using the EL2 translation regime for the Security state.
- The entry is from any level of the translation table walk.

The invalidation applies to the PE that executes this System instruction.

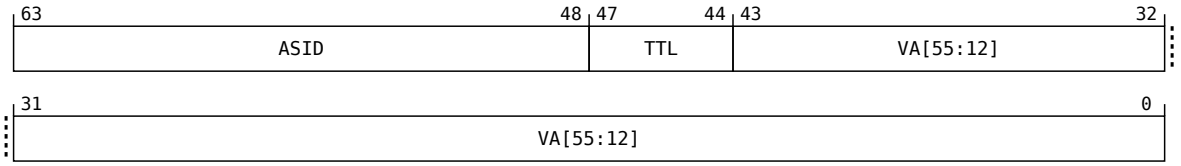
Configuration

There are no configuration notes.

Attributes

TLBI VAE2 is a 64-bit System instruction.

Field descriptions



ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
0b00xx	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
0b01xx	The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
0b10xx	The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Reserved. Treat as if TTL<3:2> is 0b00. 0b10 : Level 2. 0b11 : Level 3.
0b11xx	The entry comes from a 64KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VAE2

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`TLBI VAE2{, <Xt>}`

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0111	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH, TLBILevel_Any,
        ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.59 TLBI VAE2IS, TLB Invalidate by VA, EL2, Inner Shareable

The TLBI VAE2IS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be required to translate the specified VA using the EL2 translation regime for the Security state.
- The entry is from any level of the translation table walk.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

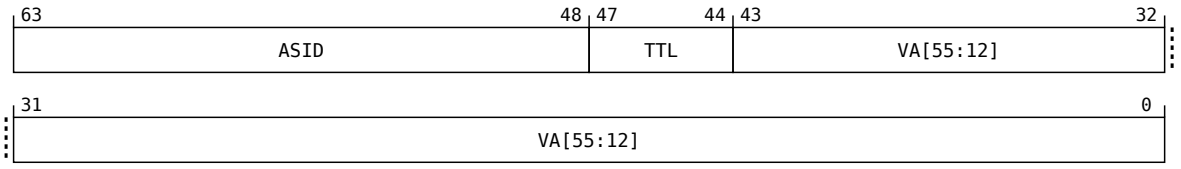
Configuration

There are no configuration notes.

Attributes

TLBI VAE2IS is a 64-bit System instruction.

Field descriptions



ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
0b00xx	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
0b01xx	The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.
0b10xx	The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Reserved. Treat as if TTL<3:2> is 0b00. 0b10 : Level 2. 0b11 : Level 3.
0b11xx	The entry comes from a 64KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VAE2IS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`TLBI VAE2IS{, <Xt>}`

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0011	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH, TLBILevel_Any,
        ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.60 TLBI VAE2OS, TLB Invalidate by VA, EL2, Outer Shareable

The TLBI VAE2OS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be required to translate the specified VA using the EL2 translation regime for the Security state.
- The entry is from any level of the translation table walk.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

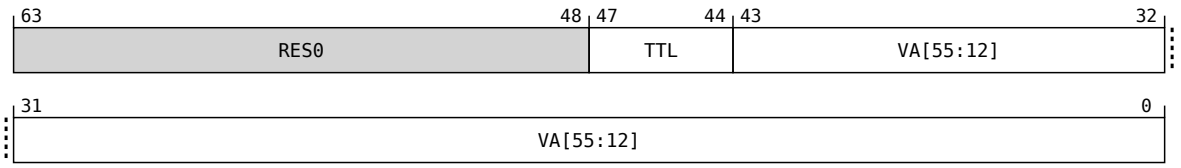
Configuration

This system instruction is present only when FEAT_TLBIOS is implemented. Otherwise, direct accesses to TLBI VAE2OS are UNDEFINED.

Attributes

TLBI VAE2OS is a 64-bit System instruction.

Field descriptions



Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
0b00xx	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
0b01xx	The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
0b10xx	The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Reserved. Treat as if TTL<3:2> is 0b00. 0b10 : Level 2. 0b11 : Level 3.
0b11xx	The entry comes from a 64KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VAE2OS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`TLBI VAE2OS{, <Xt>}`

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0001	0b001

```
if !IsFeatureImplemented(FEAT_TLBIOS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH, TLBILevel_Any,
        ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.61 TLBI VAE3, TLB Invalidate by VA, EL3

The TLBI VAE3 characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- The entry would be used to translate the specified VA using the EL3 translation regime.

The invalidation applies to the PE that executes this System instruction.

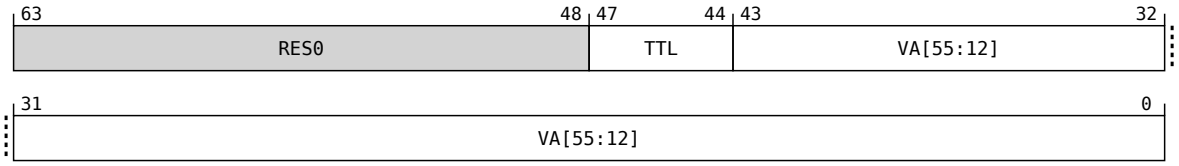
Configuration

There are no configuration notes.

Attributes

TLBI VAE3 is a 64-bit System instruction.

Field descriptions



Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
0b00xx	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
0b01xx	The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.
0b10xx	The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Reserved. Treat as if TTL<3:2> is 0b00. 0b10 : Level 2. 0b11 : Level 3.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
0b11xx	The entry comes from a 64KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VAE3

This system instruction is an alias of the SYS instruction.

This system instruction encoding is Undefined()

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VAE3{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0111	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.62 TLBI VAE3IS, TLB Invalidate by VA, EL3, Inner Shareable

The TLBI VAE3IS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- The entry would be used to translate the specified VA using the EL3 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

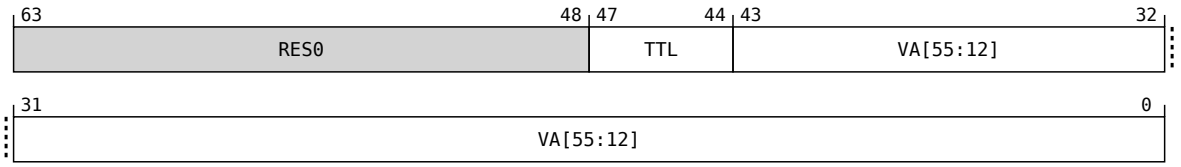
Configuration

There are no configuration notes.

Attributes

TLBI VAE3IS is a 64-bit System instruction.

Field descriptions



Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
0b00xx	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
0b01xx	The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.
0b10xx	The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Reserved. Treat as if TTL<3:2> is 0b00. 0b10 : Level 2. 0b11 : Level 3.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
0b11xx	The entry comes from a 64KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VAE3IS

This system instruction is an alias of the SYS instruction.

This system instruction encoding is Undefined()

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VAE3IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0011	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.63 TLBI VAE3OS, TLB Invalidate by VA, EL3, Outer Shareable

The TLBI VAE3OS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.
- The entry would be used to translate the specified VA using the EL3 translation regime.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

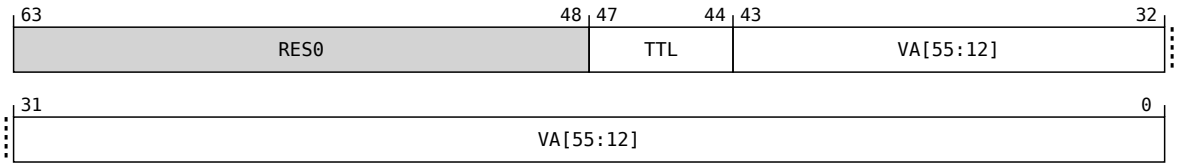
Configuration

This system instruction is present only when FEAT_TLBIOS is implemented. Otherwise, direct accesses to TLBI VAE3OS are UNDEFINED.

Attributes

TLBI VAE3OS is a 64-bit System instruction.

Field descriptions



Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TLBIOS is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
0b00xx	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
0b01xx	The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.
0b10xx	The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Reserved. Treat as if TTL<3:2> is 0b00. 0b10 : Level 2. 0b11 : Level 3.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
0b11xx	The entry comes from a 64KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VAE3OS

This system instruction is an alias of the SYS instruction.

This system instruction encoding is Undefined()

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VAE3OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0001	0b001

```
if !IsFeatureImplemented(FEAT_TLBIOS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.64 TLBI VALE1, TLB Invalidate by VA, Last level, EL1

The TLBI VALE1 characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified VA, and one of the following applies:
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.

The entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The invalidation applies to the PE that executes this System instruction.

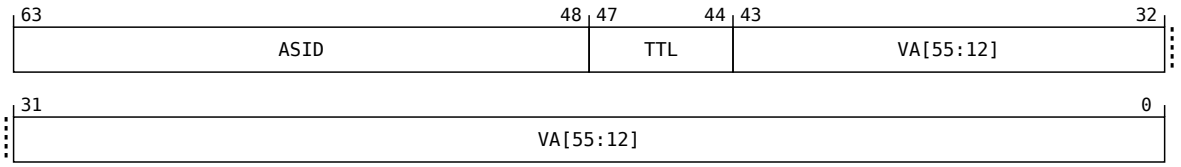
Configuration

There are no configuration notes.

Attributes

TLBI VALE1 is a 64-bit System instruction.

Field descriptions



ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
0b00xx	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
0b01xx	The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.
0b10xx	The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Reserved. Treat as if TTL<3:2> is 0b00. 0b10 : Level 2. 0b11 : Level 3.
0b11xx	The entry comes from a 64KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VALE1

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`TLBI VALE1{, <Xt>}`

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0111	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HCR_EL2.FB == '1' then
        AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
            ↪TLBILevel_Last, TLBI_AllAttr, X[t, 64]);
    else
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBILevel_Last,
        ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.65 TLBI VALE1IS, TLB Invalidate by VA, Last level, EL1, Inner Shareable

The TLBI VALE1IS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified VA, and one of the following applies:
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.

The entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

From Armv8.4, when a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, then:

- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

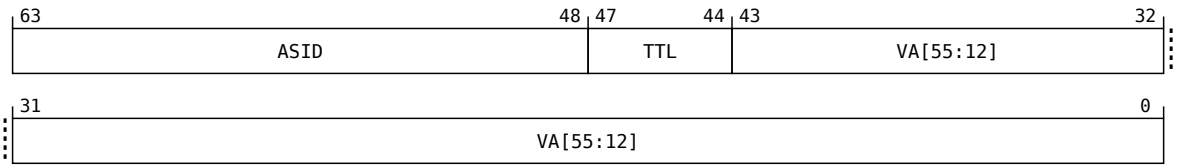
Configuration

There are no configuration notes.

Attributes

TLBI VALE1IS is a 64-bit System instruction.

Field descriptions



ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
0b00xx	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
0b01xx	The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.
0b10xx	The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Reserved. Treat as if TTL<3:2> is 0b00. 0b10 : Level 2. 0b11 : Level 3.
0b11xx	The entry comes from a 64KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VALE1IS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`TLBI VALE1IS{, <Xt>}`

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0011	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBILevel_Last,
        ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.66 TLBI VALE1OS, TLB Invalidate by VA, Last level, EL1, Outer Shareable

The TLBI VALE1OS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified VA, and one of the following applies:
  - The entry is a global entry from the final level of lookup.
  - The entry is a non-global entry from the final level of lookup that matches the specified ASID.

The entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, then:

- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

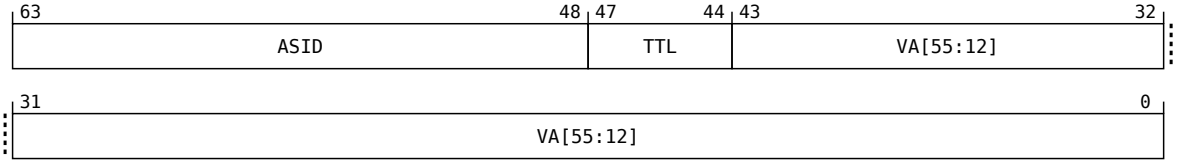
Configuration

This system instruction is present only when FEAT_TLBIOS is implemented. Otherwise, direct accesses to TLBI VALE1OS are UNDEFINED.

Attributes

TLBI VALE1OS is a 64-bit System instruction.

Field descriptions



ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
0b00xx	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
0b01xx	The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.
0b10xx	The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Reserved. Treat as if TTL<3:2> is 0b00. 0b10 : Level 2. 0b11 : Level 3.
0b11xx	The entry comes from a 64KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VALE1OS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`TLBI VALE1OS{, <Xt>}`

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0001	0b101

```
if !IsFeatureImplemented(FEAT_TLBIOS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Last,
            ↪TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_VA(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBILevel_Last,
        ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.67 TLBI VALE2, TLB Invalidate by VA, Last level, EL2

The TLBI VALE2 characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified VA using the EL2 translation regime for the Security state.
- The entry is from the final level of the translation table walk.

The invalidation applies to the PE that executes this System instruction.

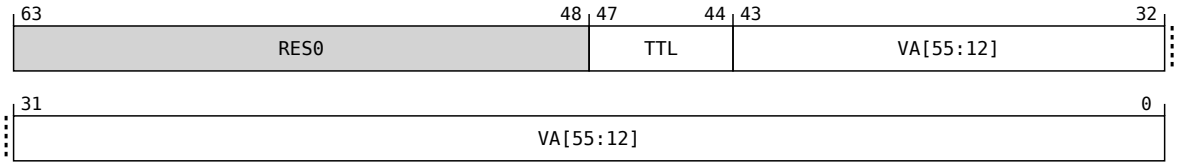
Configuration

There are no configuration notes.

Attributes

TLBI VALE2 is a 64-bit System instruction.

Field descriptions



Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
0b00xx	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
0b01xx	The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.
0b10xx	The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Reserved. Treat as if TTL<3:2> is 0b00. 0b10 : Level 2. 0b11 : Level 3.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
0b11xx	The entry comes from a 64KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VALE2

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`TLBI VALE2{, <Xt>}`

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0111	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_NSH, TLBILevel_Last,
        ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.68 TLBI VALE2IS, TLB Invalidate by VA, Last level, EL2, Inner Shareable

The TLBI VALE2IS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified VA using the EL2 translation regime for the Security state.
- The entry is from the final level of the translation table walk.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

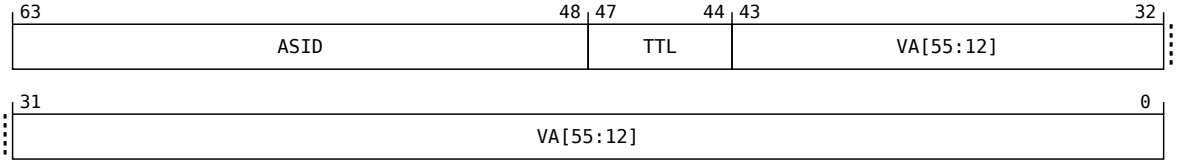
Configuration

There are no configuration notes.

Attributes

TLBI VALE2IS is a 64-bit System instruction.

Field descriptions



ASID, bits [63:48]

ASID value to match. Any TLB entries that match the ASID value and VA value will be affected by this System instruction.

Global TLB entries that match the VA value will be affected by this System instruction, regardless of the value of the ASID field.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being invalidated only uses 8 bits.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
0b00xx	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
0b01xx	The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
0b10xx	The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Reserved. Treat as if TTL<3:2> is 0b00. 0b10 : Level 2. 0b11 : Level 3.
0b11xx	The entry comes from a 64KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VALE2IS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`TLBI VALE2IS{, <Xt>}`

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0011	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_ISH, TLBILevel_Last,
        ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.69 TLBI VALE2OS, TLB Invalidate by VA, Last level, EL2, Outer Shareable

The TLBI VALE2OS characteristics are:

Purpose

When EL2 is implemented and enabled in the current Security state, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry.
- The entry would be used to translate the specified VA using the EL2 translation regime for the Security state.
- The entry is from the final level of the translation table walk.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

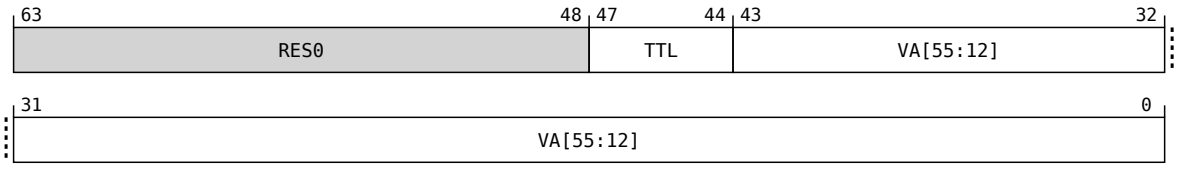
Configuration

This system instruction is present only when FEAT_TLBIOS is implemented. Otherwise, direct accesses to TLBI VALE2OS are UNDEFINED.

Attributes

TLBI VALE2OS is a 64-bit System instruction.

Field descriptions



Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
0b00xx	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
0b01xx	The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
0b10xx	The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Reserved. Treat as if TTL<3:2> is 0b00. 0b10 : Level 2. 0b11 : Level 3.
0b11xx	The entry comes from a 64KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VALE2OS

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`TLBI VALE2OS{, <Xt>}`

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0001	0b101

```
if !IsFeatureImplemented(FEAT_TLBIOS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE, Broadcast_OSH, TLBILevel_Last,
        ↪TLBI_AllAttr, X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.70 TLBI VALE3, TLB Invalidate by VA, Last level, EL3

The TLBI VALE3 characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from the final level of the translation table walk.
- The entry would be used to translate the specified VA using the EL3 translation regime.

The invalidation applies to the PE that executes this System instruction.

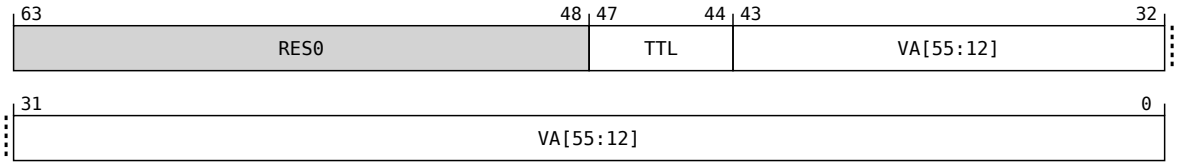
Configuration

There are no configuration notes.

Attributes

TLBI VALE3 is a 64-bit System instruction.

Field descriptions



Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
0b00xx	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
0b01xx	The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.
0b10xx	The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Reserved. Treat as if TTL<3:2> is 0b00. 0b10 : Level 2. 0b11 : Level 3.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
0b11xx	The entry comes from a 64KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VALE3

This system instruction is an alias of the SYS instruction.

This system instruction encoding is Undefined()

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VALE3{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0111	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.71 TLBI VALE3IS, TLB Invalidate by VA, Last level, EL3, Inner Shareable

The TLBI VALE3IS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from the final level of the translation table walk.
- The entry would be used to translate the specified VA using the EL3 translation regime.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

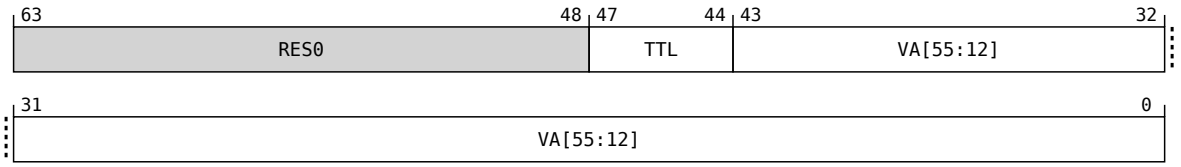
Configuration

There are no configuration notes.

Attributes

TLBI VALE3IS is a 64-bit System instruction.

Field descriptions



Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
0b00xx	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
0b01xx	The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.
0b10xx	The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Reserved. Treat as if TTL<3:2> is 0b00. 0b10 : Level 2. 0b11 : Level 3.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
0b11xx	The entry comes from a 64KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VALE3IS

This system instruction is an alias of the SYS instruction.

This system instruction encoding is Undefined()

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VALE3IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0011	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.72 TLBI VALE3OS, TLB Invalidate by VA, Last level, EL3, Outer Shareable

The TLBI VALE3OS characteristics are:

Purpose

If EL3 is implemented, invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from the final level of the translation table walk.
- The entry would be used to translate the specified VA using the EL3 translation regime.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

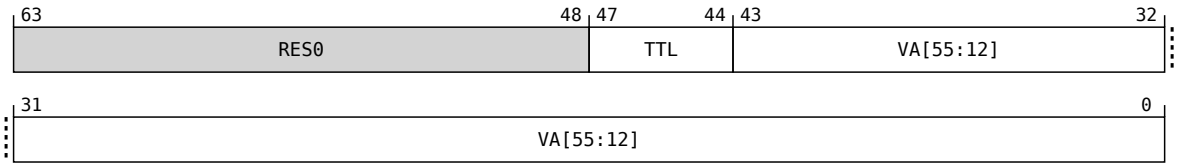
Configuration

This system instruction is present only when FEAT_TLBIOS is implemented. Otherwise, direct accesses to TLBI VALE3OS are UNDEFINED.

Attributes

TLBI VALE3OS is a 64-bit System instruction.

Field descriptions



Bits [63:48]

Reserved, RES0.

TTL, bits [47:44]

When FEAT_TTL is implemented:

Translation Table Level. Indicates the level of the translation table walk that holds the leaf entry for the address being invalidated.

TTL	Meaning
0b00xx	No information supplied as to the translation table level. Hardware must assume that the entry can be from any level. In this case, TTL<1:0> is RES0.
0b01xx	The entry comes from a 4KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.
0b10xx	The entry comes from a 16KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Reserved. Treat as if TTL<3:2> is 0b00. 0b10 : Level 2. 0b11 : Level 3.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTL	Meaning
0b11xx	The entry comes from a 64KB translation granule. The level of walk for the leaf level 0bxx is encoded as: 0b00 : Reserved. Treat as if TTL<3:2> is 0b00. 0b01 : Level 1. 0b10 : Level 2. 0b11 : Level 3.

If an incorrect value of the TTL field is specified for the entry being invalidated by the instruction, then no entries are required by the architecture to be invalidated from the TLB.

Otherwise:

Reserved, RES0.

VA[55:12], bits [43:0]

Bits[55:12] of the virtual address to match. Any appropriate TLB entries that match the ASID value (if appropriate) and VA will be affected by this System instruction.

The treatment of the low-order bits of this field depends on the translation granule size, as follows:

- Where a 4KB translation granule is being used, all bits are valid and used for the invalidation.
- Where a 16KB translation granule is being used, bits [1:0] of this field are RES0 and ignored when the instruction is executed, because VA[13:12] have no effect on the operation of the instruction.
- Where a 64KB translation granule is being used, bits [3:0] of this field are RES0 and ignored when the instruction is executed, because VA[15:12] have no effect on the operation of the instruction.

Executing TLBI VALE3OS

This system instruction is an alias of the SYS instruction.

This system instruction encoding is Undefined()

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VALE3OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b110	0b1000	0b0001	0b101

```
if !IsFeatureImplemented(FEAT_TLBIOS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.73 TLBI VMALLE1, TLB Invalidate by VMID, All at stage 1, EL1

The TLBI VMALLE1 characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.

The entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The invalidation applies to the PE that executes this System instruction.

Note

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

Configuration

There are no configuration notes.

Attributes

TLBI VMALLE1 is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing TLBI VMALLE1

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VMALLE1{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0111	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HCR_EL2.FB == '1' then
        AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ForcedISH,
            ↪TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBI_AllAttr,
            ↪X[t, 64]);
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBI_AllAttr, X[t,
        ↪64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.74 TLBI VMALLE1IS, TLB Invalidate by VMID, All at stage 1, EL1, Inner Shareable

The TLBI VMALLE1IS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.

The entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

From Armv8.4, when a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, then:

- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

Configuration

There are no configuration notes.

Attributes

TLBI VMALLE1IS is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing TLBI VMALLE1IS

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VMALLE1IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0011	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TTLB == '1' then
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBI_AllAttr,
            ↪X[t, 64]);
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBI_AllAttr, X[t,
        ↪64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.75 TLBI VMALLE1OS, TLB Invalidate by VMID, All at stage 1, EL1, Outer Shareable

The TLBI VMALLE1OS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 translation table entry, from any level of the translation table walk.

The entry would be used with the current VMID and would be required to translate the specified VA using the EL1&0 translation regime for the Security state.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, then:

- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

Configuration

This system instruction is present only when FEAT_TLBIOS is implemented. Otherwise, direct accesses to TLBI VMALLE1OS are UNDEFINED.

Attributes

TLBI VMALLE1OS is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing TLBI VMALLE1OS

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VMALLE1OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b000	0b1000	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_TLBIOS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TTLB == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBI_AllAttr,
            ↪X[t, 64]);
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_VMALL(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBI_AllAttr, X[t,
        ↪64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.76 TLBI VMALLS12E1, TLB Invalidate by VMID, All at Stage 1 and 2, EL1

The TLBI VMALLS12E1 characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 or stage 2 translation table entry, from any level of the translation table walk.
- The entry would be required to translate an address using the EL1&0 translation regime.
- The entry would be used with the current VMID.

The invalidation applies to the PE that executes this System instruction.

Note

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

Configuration

There are no configuration notes.

Attributes

TLBI VMALLS12E1 is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing TLBI VMALLS12E1

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VMALLS12E1{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0111	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_VMALLS12(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_NSH, TLBI_AllAttr, X[t,
        ↪64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.77 TLBI VMALLS12E1IS, TLB Invalidate by VMID, All at Stage 1 and 2, EL1, Inner Shareable

The TLBI VMALLS12E1IS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 or stage 2 translation table entry, from any level of the translation table walk.
- The entry would be required to translate an address using the EL1&0 translation regime.
- The entry would be used with the current VMID.

The invalidation applies to all PEs in the same Inner Shareable shareability domain as the PE that executes this System instruction.

Note

From Armv8.4, when a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, then:

- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

Configuration

There are no configuration notes.

Attributes

TLBI VMALLS12E1IS is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing TLBI VMALLS12E1IS

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VMALLS12E1IS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0011	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    AArch64.TLBI_VMALLS12(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_ISH, TLBI_AllAttr, X[t,
        ↪64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.5.78 TLBI VMALLS12E1OS, TLB Invalidate by VMID, All at Stage 1 and 2, EL1, Outer Shareable

The TLBI VMALLS12E1OS characteristics are:

Purpose

Invalidates cached copies of translation table entries from TLBs that meet all the following requirements:

- The entry is a stage 1 or stage 2 translation table entry, from any level of the translation table walk.
- The entry would be required to translate an address using the EL1&0 translation regime.
- The entry would be used with the current VMID.

The invalidation applies to all PEs in the same Outer Shareable shareability domain as the PE that executes this System instruction.

Note

When a TLB maintenance instruction is generated to the Secure EL1&0 translation regime and is defined to pass a VMID argument, then:

- A PE is architecturally required to invalidate all relevant entries in the Secure EL1&0 translation of a System MMU in the same required shareability domain with a VMID of 0.

Note

For the EL1&0 translation regimes, the invalidation applies to both global entries and non-global entries with any ASID.

Configuration

This system instruction is present only when FEAT_TLBIOS is implemented. Otherwise, direct accesses to TLBI VMALLS12E1OS are UNDEFINED.

Attributes

TLBI VMALLS12E1OS is a 64-bit System instruction.

Field descriptions

This instruction has no applicable fields.

The value in the register specified by <Xt> is ignored.

Executing TLBI VMALLS12E1OS

The Rt field should be set to `0b11111`. If the Rt field is not set to `0b11111`, it is CONSTRAINED UNPREDICTABLE whether:

- The instruction is UNDEFINED.
- The instruction behaves as if the Rt field is set to `0b11111`.

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

```
TLBI VMALLS12E1OS{, <Xt>}
```

op0	op1	CRn	CRm	op2
0b01	0b100	0b1000	0b0001	0b110

```
if !IsFeatureImplemented(FEAT_TLBIOS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    AArch64.TLBI_VMALLS12(SecurityStateAtEL(EL1), Regime_EL10, VMID[], Broadcast_OSH, TLBI_AllAttr, X[t,
        ↪64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.6 A64 System instructions for prediction restriction

This section lists the A64 System instructions for prediction restriction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.6.1 CFP RCTX, Control Flow Prediction Restriction by Context

The CFP RCTX characteristics are:

Purpose

Control Flow Prediction Restriction by Context applies to all Control Flow Prediction Resources that predict execution based on information gathered within the target execution context or contexts.

Control flow predictions determined by the actions of code in the target execution context or contexts appearing in program order before the instruction cannot exploitatively control speculative execution occurring after the instruction is complete and synchronized.

This instruction is guaranteed to be complete following a DSB that covers both read and write behavior on the same PE as executed the original restriction instruction, and a subsequent context synchronization event is required to ensure that the effect of the completion of the instructions is synchronized to the current execution.

Note

This instruction does not require the invalidation of prediction structures so long as the behavior described for completion of this instruction is met by the implementation.
On some implementations the instruction is likely to take a significant number of cycles to execute. This instruction is expected to be used very rarely, such as on the roll-over of an ASID or VMID, but should not be used on every context switch.

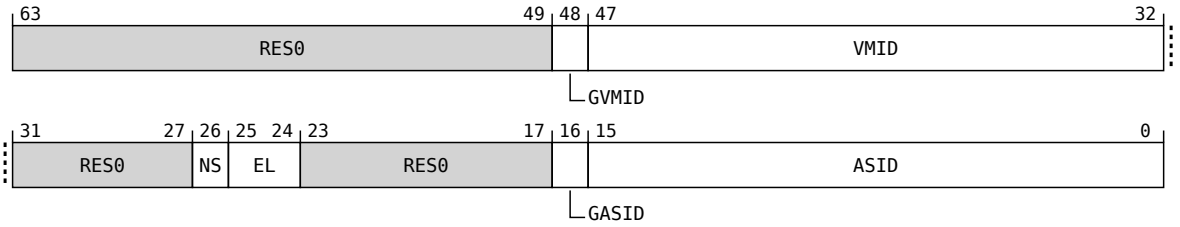
Configuration

This system instruction is present only when FEAT_SPECRES is implemented. Otherwise, direct accesses to CFP RCTX are UNDEFINED.

Attributes

CFP RCTX is a 64-bit System instruction.

Field descriptions



Bits [63:49]

Reserved, RES0.

GVMID, bit [48]

Execution of this instruction applies to all VMIDs or a specified VMID.

GVMID	Meaning
0b0	Applies to specified VMID for an EL0 or EL1 target execution context.
0b1	Applies to all VMIDs for an EL0 or EL1 target execution context.

For target execution contexts other than EL0 or EL1, this field is RES0.

If the instruction is executed at EL0 or EL1, this field has an Effective value of 0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

VMID, bits [47:32]

Only applies when bit[48] is 0 and the target execution context is either:

- EL1.
- EL0.

Otherwise this field is RES0.

When the instruction is executed at EL1, this field is treated as the current VMID.

When the instruction is executed at EL0, this field is treated as the current VMID.

If the implementation supports 16 bits of VMID, then the upper 8 bits of the VMID must be written to 0 by software when the context being affected only uses 8 bits.

Bits [31:27]

Reserved, RES0.

NS, bit [26]

Security State.

NS	Meaning
0b0	Secure state.
0b1	Non-secure state.

When executed in Non-secure state, the Effective value of NS is 1.

EL, bits [25:24]

Exception Level. Indicates the Exception level of the target execution context.

EL	Meaning
0b00	EL0.
0b01	EL1
0b10	EL2

If the instruction is executed at an Exception level lower than the specified level, or is specified to apply to a combination of Exception level and Security state that is not implemented, this instruction is treated as a NOP.

Bits [23:17]

Reserved, RES0.

GASID, bit [16]

Execution of this instruction applies to all ASIDs or a specified ASID.

GASID	Meaning
0b0	Applies to specified ASID for an EL0 target execution context.
0b1	Applies to all ASIDs for an EL0 target execution context.

For target execution contexts other than EL0, this field is RES0.

If the instruction is executed at EL0, this field has an Effective value of 0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ASID, bits [15:0]

Only applies for an EL0 target execution context and when bit[16] is 0.

Otherwise, this field is RES0.

When the instruction is executed at EL0, this field is treated as the current ASID.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being affected only uses 8 bits.

Executing CFP RCTX

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`CFP RCTX, <Xt>`

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b0011	0b100

```
if !IsFeatureImplemented(FEAT_SPECRES) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && SCTLR_EL1.EnRCTX == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        AArch64.RestrictPrediction(X[t, 64], RestrictType_ControlFlow);
elsif PSTATE.EL == EL1 then
    AArch64.RestrictPrediction(X[t, 64], RestrictType_ControlFlow);
elsif PSTATE.EL == EL2 then
    AArch64.RestrictPrediction(X[t, 64], RestrictType_ControlFlow);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.6.2 CPP RCTX, Cache Prefetch Prediction Restriction by Context

The CPP RCTX characteristics are:

Purpose

Cache Prefetch Prediction Restriction by Context applies to all Cache Allocation Resources that predict cache allocations based on information gathered within the target execution context or contexts.

The actions of code in the target execution context or contexts appearing in program order before the instruction cannot exploitatively control cache prefetch predictions occurring after the instruction is complete and synchronized.

This instruction applies to all:

- Instruction caches.
- Data caches.
- TLB prefetching hardware used by the executing PE that applies to the supplied context or contexts.

This instruction is guaranteed to be complete following a DSB that covers both read and write behavior on the same PE as executed the original restriction instruction, and a subsequent context synchronization event is required to ensure that the effect of the completion of the instructions is synchronized to the current execution.

Note

This instruction does not require the invalidation of Cache Allocation Resources so long as the behavior described for completion of this instruction is met by the implementation.
On some implementations the instruction is likely to take a significant number of cycles to execute. This instruction is expected to be used very rarely, such as on the roll-over of an ASID or VMID, but should not be used on every context switch.

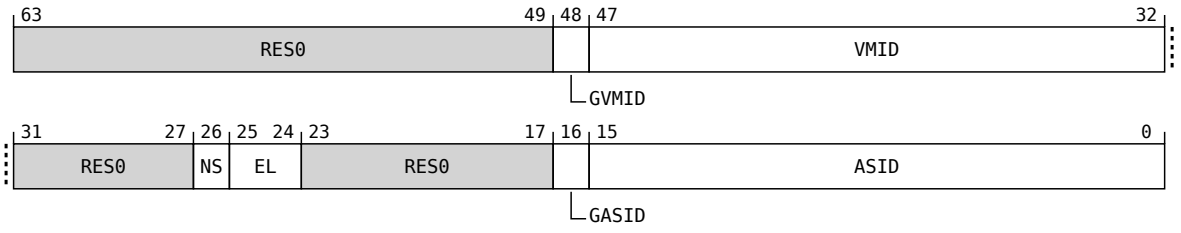
Configuration

This system instruction is present only when FEAT_SPECRES is implemented. Otherwise, direct accesses to CPP RCTX are UNDEFINED.

Attributes

CPP RCTX is a 64-bit System instruction.

Field descriptions



Bits [63:49]

Reserved, RES0.

GVMID, bit [48]

Execution of this instruction applies to all VMIDs or a specified VMID.

GVMID	Meaning
0b0	Applies to specified VMID for an EL0 or EL1 target execution context.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

GVMID	Meaning
0b1	Applies to all VMIDs for an EL0 or EL1 target execution context.

For target execution contexts other than EL0 and EL1, this field is RES0.

If the instruction is executed at EL0 or EL1, this field has an Effective value of 0.

VMID, bits [47:32]

Only applies when bit[48] is 0 and the target execution context is either:

- EL1.
- EL0.

Otherwise this field is RES0.

When the instruction is executed at EL1, this field is treated as the current VMID.

When the instruction is executed at EL0, this field is treated as the current VMID.

If the implementation supports 16 bits of VMID, then the upper 8 bits of the VMID must be written to 0 by software when the context being affected only uses 8 bits.

Bits [31:27]

Reserved, RES0.

NS, bit [26]

Security State.

NS	Meaning
0b0	Secure state.
0b1	Non-secure state.

When executed in Non-secure state, the Effective value of NS is 1.

EL, bits [25:24]

Exception Level. Indicates the Exception level of the target execution context.

EL	Meaning
0b00	EL0.
0b01	EL1
0b10	EL2

If the instruction is executed at an Exception level lower than the specified level, or is specified to apply to a combination of Exception level and Security state that is not implemented, this instruction is treated as a NOP.

Bits [23:17]

Reserved, RES0.

GASID, bit [16]

Execution of this instruction applies to all ASIDs or a specified ASID.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

GASID	Meaning
0b0	Applies to specified ASID for an EL0 target execution context.
0b1	Applies to all ASIDs for an EL0 target execution context.

For target execution contexts other than EL0, this field is RES0.

If the instruction is executed at EL0, this field has an Effective value of 0.

ASID, bits [15:0]

Only applies for an EL0 target execution context and when bit[16] is 0.

Otherwise, this field is RES0.

When the instruction is executed at EL0, this field is treated as the current ASID.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being affected only uses 8 bits.

Executing CPP RCTX

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`CPP RCTX, <Xt>`

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b0011	0b111

```
if !IsFeatureImplemented(FEAT_SPECRES) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && SCTLR_EL1.EnRCTX == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        AArch64.RestrictPrediction(X[t, 64], RestrictType_CachePrefetch);
elsif PSTATE.EL == EL1 then
    AArch64.RestrictPrediction(X[t, 64], RestrictType_CachePrefetch);
elsif PSTATE.EL == EL2 then
    AArch64.RestrictPrediction(X[t, 64], RestrictType_CachePrefetch);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.6.3 DVP RCTX, Data Value Prediction Restriction by Context

The DVP RCTX characteristics are:

Purpose

Data Value Prediction Restriction by Context applies to all Data Value Prediction Resources that predict execution based on information gathered within the target execution context or contexts.

Note

The prediction of the PSTATE.{N,Z,C,V} values is not considered a data value for this purpose.

Data value predictions determined by the actions of code in the target execution context or contexts appearing in program order before the instruction cannot exploitatively control speculative execution occurring after the instruction is complete and synchronized.

This instruction is guaranteed to be complete following a DSB that covers both read and write behavior on the same PE as executed the original restriction instruction, and a subsequent context synchronization event is required to ensure that the effect of the completion of the instructions is synchronized to the current execution.

Note

This instruction does not require the invalidation of prediction structures so long as the behavior described for completion of this instruction is met by the implementation.
On some implementations the instruction is likely to take a significant number of cycles to execute. This instruction is expected to be used very rarely, such as on the roll-over of an ASID or VMID, but should not be used on every context switch.

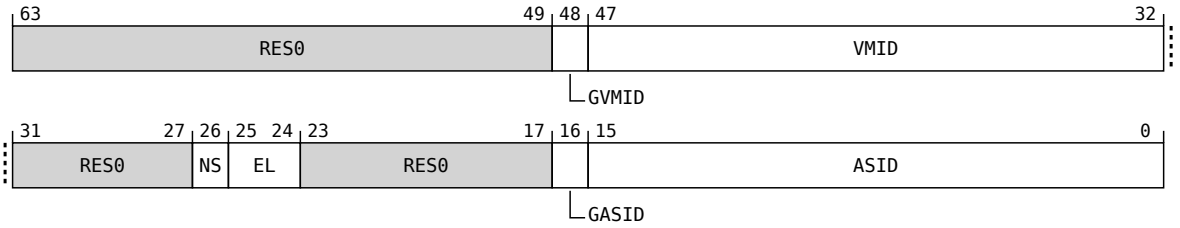
Configuration

This system instruction is present only when FEAT_SPECRES is implemented. Otherwise, direct accesses to DVP RCTX are UNDEFINED.

Attributes

DVP RCTX is a 64-bit System instruction.

Field descriptions



Bits [63:49]

Reserved, RES0.

GVMID, bit [48]

Execution of this instruction applies to all VMIDs or a specified VMID.

GVMID	Meaning
0b0	Applies to specified VMID for an EL0 or EL1 target execution context.
0b1	Applies to all VMIDs for an EL0 or EL1 target execution context.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For target execution contexts other than EL0 or EL1, this field is RES0.

If the instruction is executed at EL0 or EL1, then this field has an Effective value of 0.

VMID, bits [47:32]

Only applies when bit[48] is 0 and the target execution context is either:

- EL1.
- EL0.

Otherwise this field is RES0.

When the instruction is executed at EL1, this field is treated as the current VMID.

When the instruction is executed at EL0, this field is treated as the current VMID.

If the implementation supports 16 bits of VMID, then the upper 8 bits of the VMID must be written to 0 by software when the context being affected only uses 8 bits.

Bits [31:27]

Reserved, RES0.

NS, bit [26]

Security State.

NS	Meaning
0b0	Secure state.
0b1	Non-secure state.

When executed in Non-secure state, the Effective value of NS is 1.

EL, bits [25:24]

Exception Level. Indicates the Exception level of the target execution context.

EL	Meaning
0b00	EL0.
0b01	EL1
0b10	EL2

If the instruction is executed at an Exception level lower than the specified level, or is specified to apply to a combination of Exception level and Security state that is not implemented, this instruction is treated as a NOP.

Bits [23:17]

Reserved, RES0.

GASID, bit [16]

Execution of this instruction applies to all ASIDs or a specified ASID.

GASID	Meaning
0b0	Applies to specified ASID for an EL0 target execution context.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b1	Applies to all ASIDs for an EL0 target execution context.

For target execution contexts other than EL0, this field is RES0.

If the instruction is executed at EL0, this field has an Effective value of 0.

ASID, bits [15:0]

Only applies for an EL0 target execution context and when bit[16] is 0.

Otherwise this field is RES0.

When the instruction is executed at EL0, this field is treated as the current ASID.

If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being affected only uses 8 bits.

Executing DVP RCTX

This system instruction is an alias of the SYS instruction.

Accesses to this instruction use the following encodings in the System instruction encoding space:

`DVP RCTX, <Xt>`

op0	op1	CRn	CRm	op2
0b01	0b011	0b0111	0b0011	0b101

```
if !IsFeatureImplemented(FEAT_SPECRES) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && SCTLR_EL1.EnRCTX == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        AArch64.RestrictPrediction(X[t, 64], RestrictType_DataValue);
elsif PSTATE.EL == EL1 then
    AArch64.RestrictPrediction(X[t, 64], RestrictType_DataValue);
elsif PSTATE.EL == EL2 then
    AArch64.RestrictPrediction(X[t, 64], RestrictType_DataValue);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.7 A64 IMPLEMENTATION DEFINED System instructions

This section lists the A64 IMPLEMENTATION DEFINED System instructions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C5.7.1 SYS|SYSL S1_<op1>_<Cn>_<Cm>_<op2>, IMPLEMENTATION DEFINED System instructions

The SYS|SYSL S1_<op1><*Cn*><Cm>_<op2> characteristics are:

Purpose

This area of the System instruction encoding space is reserved for IMPLEMENTATION DEFINED System instructions.

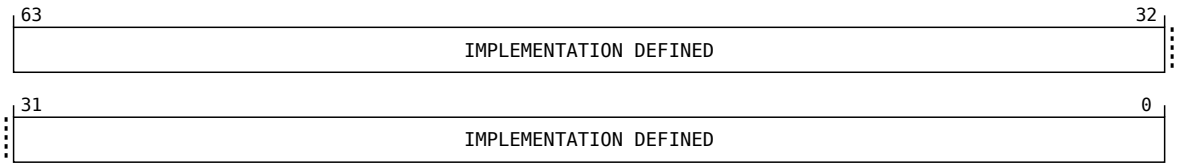
Configuration

There are no configuration notes.

Attributes

SYS|SYSL S1_<op1><*Cn*><Cm>_<op2> is a 64-bit System instruction.

Field descriptions



IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

Executing SYS|SYSL S1_<op1>_<Cn>_<Cm>_<op2>

Accesses to this instruction use the following encodings in the System instruction encoding space:

`SYS #<op1>, <Cn>, <Cm>, #<op2>{, <Xt>}`

op0	op1	CRn	CRm	op2
0b01	op1[2:0]	0b1x11	Cm[3:0]	op2[2:0]

```
if PSTATE.EL == EL0 then
    AArch64.ImpDefSysInstr(op0, op1, CRn, CRm, op2, t);
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TIDCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.ImpDefSysInstr(op0, op1, CRn, CRm, op2, t);
elsif PSTATE.EL == EL2 then
    AArch64.ImpDefSysInstr(op0, op1, CRn, CRm, op2, t);
```

`SYSL <Xt>, #<op1>, <Cn>, <Cm>, #<op2>`

op0	op1	CRn	CRm	op2
0b01	op1[2:0]	0b1x11	Cm[3:0]	op2[2:0]

```
if PSTATE.EL == EL0 then
    AArch64.ImpDefSysInstrWithResult(op0, op1, CRn, CRm, op2, t);
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TIDCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        AArch64.ImpDefSysInstrWithResult(op0, op1, CRn, CRm, op2, t);
elsif PSTATE.EL == EL2 then
    AArch64.ImpDefSysInstrWithResult(op0, op1, CRn, CRm, op2, t);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter C6
A64 Base Instruction Descriptions

This chapter describes the A64 base instructions.

It contains the following sections:

- About the A64 base instructions.
- Alphabetical list of A64 base instructions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.1 About the A64 base instructions

Alphabetical list of A64 base instructions gives full descriptions of the A64 instructions that are in the following instruction groups:

- Branch, Exception generation, and System instructions.
- Loads and stores associated with the general-purpose registers.
- Data processing (immediate).
- Data processing (register).

A64 instruction set encoding provides an overview of the instruction encodings as well as of the instruction classes within their functional groups.

The rest of this section is general description of the base instructions. It contains the following subsections:

- Register size.
- Use of the PC.
- Use of the stack pointer.
- Condition flags and related instructions.

C6.1.1 Register size

Most data processing, comparison, and conversion instructions that use the general-purpose registers as the source or destination operand have two instruction variants that operate on either a 32-bit or a 64-bit value.

Where a 32-bit instruction form is selected, the following holds:

- The upper 32 bits of the source registers are ignored.
- The upper 32 bits of the destination register are set to zero.
- Right shifts and right rotates inject at bit[31], not at bit[63].
- The Condition flags, where set by the instruction, are computed from the lower 32 bits.

This distinction applies even when the results of a 32-bit instruction form are indistinguishable from the lower 32 bits computed by the equivalent 64-bit instruction form. For example, a 32-bit bitwise `ORR` could be performed using a 64-bit `ORR` and simply ignoring the top 32 bits of the result. However, the A64 instruction set includes separate 32-bit and 64-bit forms of the `ORR` instruction.

As well as distinct sign-extend or zero-extend instructions, the A64 instruction set also provides the ability to extend and shift the final source register of an `ADD`, `SUB`, `ADDS`, `or SUBS` instruction and the index register of a load/store instruction. This enables array index calculations involving a 64-bit array pointer and a 32-bit array index to be implemented efficiently.

The assembly language notation enables the distinct identification of registers holding 32-bit values and registers holding 64-bit values. See Register names and Register indexed addressing.

C6.1.2 Use of the PC

A64 instructions have limited access to the PC. The only instructions that can read the PC are those that generate a PC relative address:

- ADR and ADRP.
- The Load register (literal) instruction class.
- Direct branches that use an immediate offset.
- The unconditional branch with link instructions, BL and BLR, that use the PC to create the return link address.

Only explicit control flow instructions can modify the PC:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Conditional and unconditional branch and return instructions.
- Exception generation and exception return instructions.

For more details of instructions that can modify the PC, see Branches, Exception generating, and System instructions.

C6.1.3 Use of the stack pointer

A64 instructions can use the stack pointer only in a limited number of cases:

- Load/store instructions use the current stack pointer as the base address:
  - When stack alignment checking is enabled by system software and the base register is SP, the current stack pointer must be initially quadword aligned, That is, it must be aligned to 16 bytes. Misalignment generates an SP alignment fault. See SP alignment checking for more information.
- Add and subtract data processing instructions in their immediate and extended register forms, use the current stack pointer as a source register or the destination register or both.
- Logical data processing instructions in their immediate form use the current stack pointer as the destination register.

C6.1.4 Condition flags and related instructions

The A64 base instructions that use the Condition flags as an input are:

- Conditional branch. The conditional branch instruction is `B.cond`.
- Add or subtract with carry. These instruction types include instructions to perform multi-precision arithmetic and calculate checksums. The add or subtract with carry instructions are `ADC`, `ADCS`, `SBC`, and `SBCS`, or an architectural alias for these instructions.
- Conditional select with increment, negate, or invert. This instruction type conditionally selects between one source register and a second, incremented, negated, inverted, or unmodified source register. The conditional select with increment, negate, or invert instructions are `CSINC`, `CSINV`, and `CSNEG`.

  These instructions also implement:
  - Conditional select or move. The Condition flags select one of two source registers as the destination register. Short conditional sequences can be replaced by unconditional instructions followed by a conditional select, `CSEL`.
  - Conditional set. Conditionally selects between 0 and 1, or 0 and -1. This can be used to convert the Condition flags to a Boolean value or mask in a general-purpose register, for example. These instructions include `CSET` and `CSETM`.
- Conditional compare. This instruction type sets the Condition flags to the result of a comparison if the original condition is true, otherwise it sets the Condition flags to an immediate value. It permits the flattening of nested conditional expressions without using conditional branches or performing Boolean arithmetic within the general-purpose registers. The conditional compare instructions are `CCMP` and `CCMN`.

The A64 base instructions that update the Condition flags as an output are:

- Flag-setting data processing instructions, such as `ADCS`, `ADDS`, `ANDS`, `BICS`, `RMIF`, `SBCS`, `SETF8`,`SETF16`, and `SUBS`, and the aliases `CMN`, `CMP`, and `TST`.
- Conditional compare instructions such as `CCMN`, `CCMP`.
- The random number generation instructions `MRS RNDR` and `MRS RNDRRS`, see Effect of random number generation instructions on Condition flags.

The A64 base instructions that manipulate the Condition flags are:

- The flag manipulation instruction `CFINV`, which inverts the value of the Carry flag.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If FEAT_FlagM2 is implemented, the base instructions `AXFLAG` and `XAFLAG`. These instructions convert between the Arm floating point comparison PSTATE condition flag format and an alternative format shown in Table C6-1.

Table C6-1 Relationship between ARM format and alternative format PSTATE condition flags

	ARM format				Alternative format			
Result	N	Z	C	V	N	Z	C	V
Greater than	0	0	1	0	0	0	1	0
Less than	1	0	0	0	0	0	0	0
Equal	0	1	1	0	0	1	1	0
Unordered	0	0	1	1	0	1	0	0

The flags can be directly accessed for a read/write using the NZCV.

The A64 base instructions also include conditional branch instructions that do not manipulate the Condition flags:

- Compare and branch if a register is zero or nonzero, `CBZ` and `CBNZ`.
- Test a single bit in a register and branch if the bit is zero or nonzero, `TBZ` and `TBNZ`.
- Compare and branch instructions, `CB<cc>` (immediate), `CB<cc>` (register), `CBB<cc>`, and `CBH<cc>`.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2 Alphabetical list of A64 base instructions

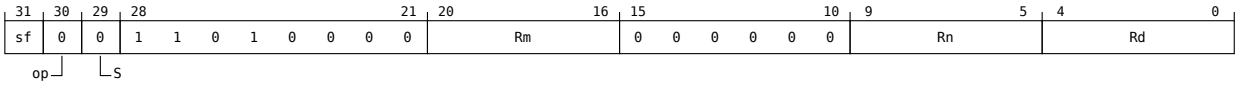
This section lists every instruction in the base category of the A64 instruction set. For details of the format used, see Understanding the A64 instruction descriptions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.1 ADC

Add with carry

This instruction adds two register values and the Carry flag value, and writes the result to the destination register.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
ADC <Wd>, <Wn>, <Wm>
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
ADC <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
```

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wn**> Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Wm**> Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Xn**> Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Xm**> Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Operation

```
constant bits(datasize) operand1 = X[n, datasize];
constant bits(datasize) operand2 = X[m, datasize];
bits(datasize) result;

(result, -) = AddWithCarry(operand1, operand2, PSTATE.C);

X[d, datasize] = result;
```

Operational information

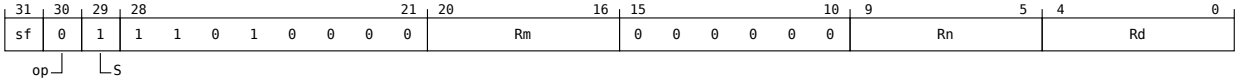
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.2 ADCS

Add with carry, setting flags

This instruction adds two register values and the Carry flag value, and writes the result to the destination register. It updates the condition flags based on the result.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
ADCS <Wd>, <Wn>, <Wm>
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
ADCS <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
```

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wn**> Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Wm**> Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Xn**> Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Xm**> Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Operation

```
constant bits(datasize) operand1 = X[n, datasize];
constant bits(datasize) operand2 = X[m, datasize];
bits(datasize) result;
bits(4) nzcv;

(result, nzcv) = AddWithCarry(operand1, operand2, PSTATE.C);

X[d, datasize] = result;
PSTATE.<N,Z,C,V> = nzcv;
```

Operational information

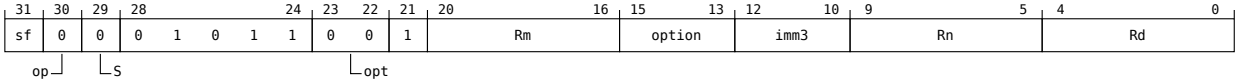
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.3 ADD (extended register)

Add extended and scaled register

This instruction adds a register value and a sign or zero-extended register value, followed by an optional left shift amount, and writes the result to the destination register. The argument that is extended from the <Rm> register can be a byte, halfword, word, or doubleword.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
ADD <Wd|WSP>, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
ADD <Xd|SP>, <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

Decode for all variants of this encoding

```
if imm3 IN {'101', '110', '111'} then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer shift = UInt(imm3);
constant integer datasize = 32 << UInt(sf);
constant ExtendType extend_type = DecodeRegExtend(option);
```

Assembler Symbols

<**Wd**|**WSP**> Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

<**Wn**|**WSP**> Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<**Wm**> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<**extend**> For the "32-bit" variant: is the extension to be applied to the second source operand, encoded in "option":

option	<extend>
000	UXTB
001	UXTH
010	LSL|UXTW
011	UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

If "Rd" or "Rn" is '11111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is '010'.

For the "64-bit" variant: is the extension to be applied to the second source operand, encoded in "option":

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

option	<extend>
000	UXTB
001	UXTH
010	UXTW
011	LSL|UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

If “Rd” or “Rn” is ’11111’ (SP) and “option” is ’011’ then LSL is preferred, but may be omitted when “imm3” is ’000’. In all other cases <extend> is required and must be UXTX when “option” is ’011’.

<**amount**> Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the “imm3” field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

<**Xd|SP**> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.

<**Xn|SP**> Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the “Rn” field.

<**R**> Is a width specifier, encoded in “option”:

option	<R>
00x	W
010	W
x11	X
10x	W
110	W

<**m**> Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the “Rm” field.

Operation

```
constant bits(datasize) operand1 = if n == 31 then SP[datasize] else X[n, datasize];
constant bits(datasize) operand2 = ExtendReg(m, extend_type, shift, datasize);
bits(datasize) result;

(result, -) = AddWithCarry(operand1, operand2, '0');

if d == 31 then
    SP[64] = ZeroExtend(result, 64);
else
    X[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

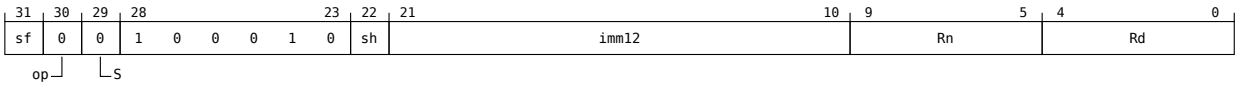
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.4 ADD (immediate)

Add immediate value

This instruction adds a register value and an optionally-shifted immediate value, and writes the result to the destination register.

This instruction is used by the alias MOV (to/from SP).



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
ADD <Wd|WSP>, <Wn|WSP>, #<imm>{, <shift>}
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
ADD <Xd|SP>, <Xn|SP>, #<imm>{, <shift>}
```

Decode for all variants of this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);

constant bits(24) imm = if sh == '0' then Zeros(12):imm12 else imm12:Zeros(12);
```

Assembler Symbols

<**Wd**|**WSP**> Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.

<**Wn**|**WSP**> Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the “Rn” field.

<**imm**> Is an unsigned immediate, in the range 0 to 4095, encoded in the “imm12” field.

<**shift**> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

sh	<shift>
0	LSL #0
1	LSL #12

<**Xd**|**SP**> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.

<**Xn**|**SP**> Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the “Rn” field.

Alias Conditions

Alias	Is preferred when
MOV (to/from SP)	`sh == '0' && imm12 == '000000000000' && (Rd == '11111' || Rn == '11111')`

Operation

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant bits(datasize) operand1 = if n == 31 then SP[datasize] else X[n, datasize];
constant bits(datasize) operand2 = ZeroExtend(imm, datasize);
bits(datasize) result;
(result, -) = AddWithCarry(operand1, operand2, '0');

if d == 31 then
    SP[64] = ZeroExtend(result, 64);
else
    X[d, datasize] = result;
```

Operational information

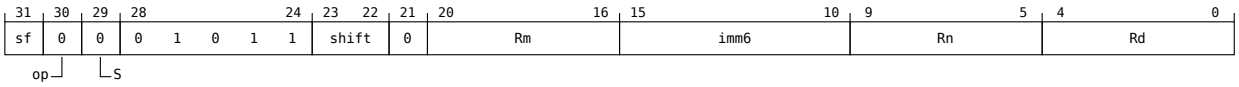
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.5 ADD (shifted register)

Add optionally-shifted register

This instruction adds a register value and an optionally-shifted register value, and writes the result to the destination register.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
ADD <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
ADD <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

Decode for all variants of this encoding

```
if shift == '11' then EndOfDecode(Decode_UNDEF);
if sf == '0' && imm6<5> == '1' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
constant ShiftType shift_type = DecodeShift(shift);
constant integer shift_amount = UInt(imm6);
```

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wn**> Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Wm**> Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<**shift**> Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in “shift”:

shift	<shift>
00	LSL
01	LSR
10	ASR
11	RESERVED

<**amount**> For the “32-bit” variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm6” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Xn**> Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Xm**> Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
constant bits(datasize) operand1 = X[n, datasize];
constant bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount, datasize);
bits(datasize) result;

(result, -) = AddWithCarry(operand1, operand2, '0');

X[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

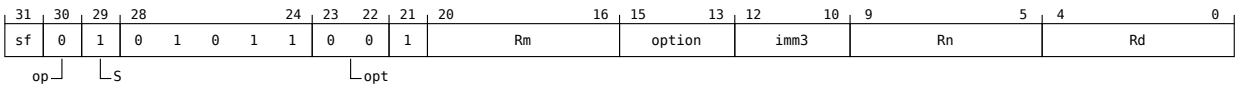
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.6 ADDS (extended register)

Add extended and scaled register, setting flags

This instruction adds a register value and a sign or zero-extended register value, followed by an optional left shift amount, and writes the result to the destination register. The argument that is extended from the <Rm> register can be a byte, halfword, word, or doubleword. It updates the condition flags based on the result.

This instruction is used by the alias CMN (extended register).



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
ADDS <Wd>, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
ADDS <Xd>, <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

Decode for all variants of this encoding

```
if imm3 IN {'101', '110', '111'} then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer shift = UInt(imm3);
constant integer datasize = 32 << UInt(sf);
constant ExtendType extend_type = DecodeRegExtend(option);
```

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn|WSP> Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<extend> For the "32-bit" variant: is the extension to be applied to the second source operand, encoded in "option":

option	<extend>
000	UXTB
001	UXTH
010	LSL|UXTW
011	UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

If "Rn" is '11111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is '010'.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For the “64-bit” variant: is the extension to be applied to the second source operand, encoded in “option”:

option	<extend>
000	UXTB
001	UXTH
010	UXTW
011	LSL|UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

If “Rn” is ’11111’ (SP) and “option” is ’011’ then LSL is preferred, but may be omitted when “imm3” is ’000’. In all other cases <extend> is required and must be UXTX when “option” is ’011’.

<**amount**> Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the “imm3” field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Xn|SP**> Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the “Rn” field.

<**R**> Is a width specifier, encoded in “option”:

option	<R>
00x	W
010	W
x11	X
10x	W
110	W

<**m**> Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the “Rm” field.

Alias Conditions

Alias	Is preferred when
CMN (extended register)	Rd == '11111'

Operation

```
constant bits(datasize) operand1 = if n == 31 then SP[datasize] else X[n, datasize];
constant bits(datasize) operand2 = ExtendReg(m, extend_type, shift, datasize);
bits(datasize) result;
bits(4) nzcv;

(result, nzcv) = AddWithCarry(operand1, operand2, '0');

X[d, datasize] = result;
PSTATE.<N,Z,C,V> = nzcv;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

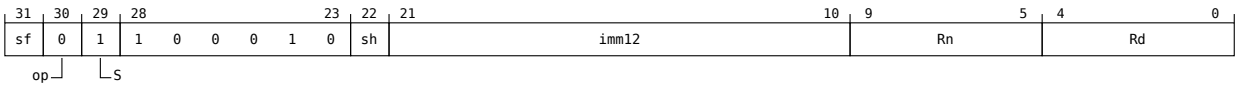
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.7 ADDS (immediate)

Add immediate value, setting flags

This instruction adds a register value and an optionally-shifted immediate value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias CMN (immediate).



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
ADDS <Wd>, <Wn|WSP>, #<imm>{, <shift>}
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
ADDS <Xd>, <Xn|SP>, #<imm>{, <shift>}
```

Decode for all variants of this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);

constant bits(24) imm = if sh == '0' then Zeros(12):imm12 else imm12:Zeros(12);
```

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Wn|WSP**> Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

<**imm**> Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.

<**shift**> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in "sh":

sh	<shift>
0	LSL #0
1	LSL #12

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Xn|SP**> Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
CMN (immediate)	Rd == '11111'

Operation

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant bits(datasize) operand1 = if n == 31 then SP[datasize] else X[n, datasize];
constant bits(datasize) operand2 = ZeroExtend(imm, datasize);
bits(datasize) result;
bits(4) nzcv;

(result, nzcv) = AddWithCarry(operand1, operand2, '0');

X[d, datasize] = result;
PSTATE.<N,Z,C,V> = nzcv;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

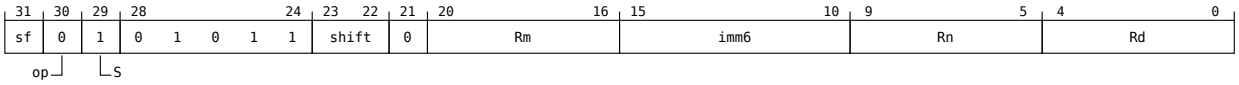
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.8 ADDS (shifted register)

Add optionally-shifted register, setting flags

This instruction adds a register value and an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias CMN (shifted register).



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
ADDS <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
ADDS <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

Decode for all variants of this encoding

```
if shift == '11' then EndOfDecode(Decode_UNDEF);
if sf == '0' && imm6<5> == '1' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
constant ShiftType shift_type = DecodeShift(shift);
constant integer shift_amount = UInt(imm6);
```

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Wn**> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

<**Wm**> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<**shift**> Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in "shift":

shift	<shift>
00	LSL
01	LSR
10	ASR
11	RESERVED

<**amount**> For the "32-bit" variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the "64-bit" variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Xn**> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**Xm**> Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Alias Conditions

Alias	Is preferred when
CMN (shifted register)	Rd == '11111'

Operation

```
constant bits(datasize) operand1 = X[n, datasize];
constant bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount, datasize);
bits(datasize) result;
bits(4) nzcv;

(result, nzcv) = AddWithCarry(operand1, operand2, '0');

X[d, datasize] = result;
PSTATE.<N,Z,C,V> = nzcv;
```

Operational information

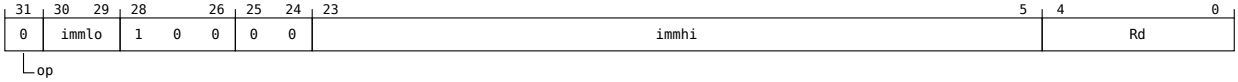
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.9 ADR

Form PC-relative address

This instruction adds an immediate value to the PC value to form a PC-relative address, and writes the result to the destination register.



Encoding

```
ADR <Xd>, <label>
```

Decode for this encoding

```
constant integer d = UInt(Rd);
constant bits(64) imm = SignExtend(immhi:immlo, 64);
```

Assembler Symbols

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**label**> Is the program label whose address is to be calculated. Its offset from the address of this instruction, in the range +/-1MB, is encoded in “immhi:immlo”.

Operation

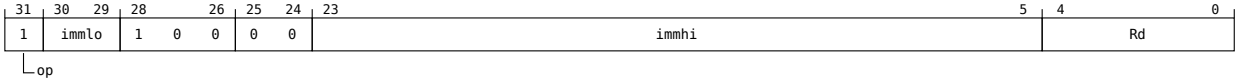
```
X[d, 64] = PC64 + imm;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.10 ADRP

Form PC-relative address to 4KB page

This instruction adds an immediate value that is shifted left by 12 bits, to the PC value to form a PC-relative address, with the bottom 12 bits masked out, and writes the result to the destination register.



Encoding

```
ADRP <Xd>, <label>
```

Decode for this encoding

```
constant integer d = UInt(Rd);
constant bits(64) imm = SignExtend(immhi:immlo:Zeros(12), 64);
```

Assembler Symbols

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**label**> Is the program label whose 4KB page address is to be calculated. Its offset from the page address of this instruction, in the range +/-4GB, is encoded as "immhi:immlo" times 4096.

Operation

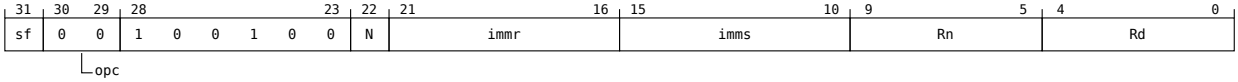
```
constant bits(64) base = PC64<63:12>:Zeros(12);
X[d, 64] = base + imm;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.11 AND (immediate)

Bitwise AND (immediate)

This instruction performs a bitwise AND of a register value and an immediate value, and writes the result to the destination register.



Encoding for the 32-bit variant Applies when `(sf == 0 && N == 0)`

```
AND <Wd|WSP>, <Wn>, #<imm>
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
AND <Xd|SP>, <Xn>, #<imm>
```

Decode for all variants of this encoding

```
if sf == '0' && N != '0' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);
bits(datasize) imm;
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE, datasize);
```

Assembler Symbols

<**Wd|WSP**> Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

<**Wn**> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

<**imm**> For the "32-bit" variant: is the bitmask immediate, encoded in "imms:immr".

For the "64-bit" variant: is the bitmask immediate, encoded in "N:imms:immr".

<**Xd|SP**> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

<**Xn**> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

```
constant bits(datasize) operand1 = X[n, datasize];
constant bits(datasize) operand2 = imm;

constant bits(datasize) result = operand1 AND operand2;

if d == 31 then
    SP[64] = ZeroExtend(result, 64);
else
    X[d, datasize] = result;
```

Operational information

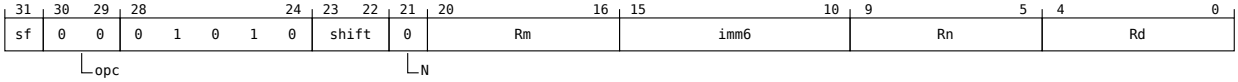
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.12 AND (shifted register)

Bitwise AND (shifted register)

This instruction performs a bitwise AND of a register value and an optionally-shifted register value, and writes the result to the destination register.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
AND <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
AND <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

Decode for all variants of this encoding

```
if sf == '0' && imm6<5> == '1' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
constant ShiftType shift_type = DecodeShift(shift);
constant integer shift_amount = UInt(imm6);
```

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wn**> Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Wm**> Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<**shift**> Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:

shift	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<**amount**> For the “32-bit” variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm6” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Xn**> Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Xm**> Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
constant bits(datasize) operand1 = X[n, datasize];
constant bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount, datasize);

X[d, datasize] = operand1 AND operand2;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

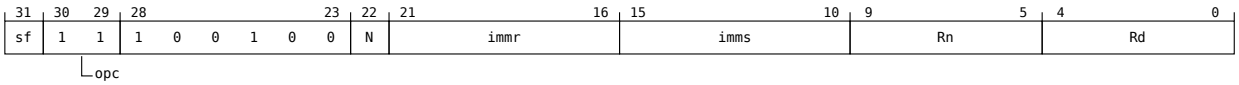
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.13 ANDS (immediate)

Bitwise AND (immediate), setting flags

This instruction performs a bitwise AND of a register value and an immediate value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias TST (immediate).



Encoding for the 32-bit variant Applies when `(sf == 0 && N == 0)`

```
ANDS <Wd>, <Wn>, #<imm>
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
ANDS <Xd>, <Xn>, #<imm>
```

Decode for all variants of this encoding

```
if sf == '0' && N != '0' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);
bits(datasize) imm;
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE, datasize);
```

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wn**> Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<**imm**> For the “32-bit” variant: is the bitmask immediate, encoded in “imms:immr”.

For the “64-bit” variant: is the bitmask immediate, encoded in “N:imms:immr”.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Xn**> Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Alias Conditions

Alias	Is preferred when
TST (immediate)	`Rd == '11111'`

Operation

```
constant bits(datasize) operand1 = X[n, datasize];
constant bits(datasize) operand2 = imm;

constant bits(datasize) result = operand1 AND operand2;

X[d, datasize] = result;
PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

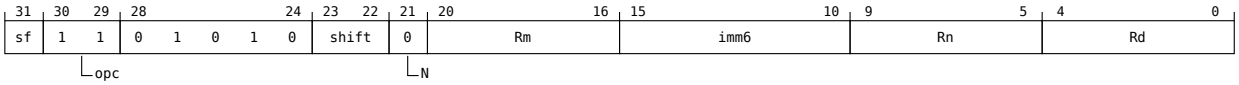
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.14 ANDS (shifted register)

Bitwise AND (shifted register), setting flags

This instruction performs a bitwise AND of a register value and an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias TST (shifted register).



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
ANDS <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
ANDS <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

Decode for all variants of this encoding

```
if sf == '0' && imm6<5> == '1' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
constant ShiftType shift_type = DecodeShift(shift);
constant integer shift_amount = UInt(imm6);
```

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wn**> Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Wm**> Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<**shift**> Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:

shift	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<**amount**> For the “32-bit” variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm6” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Xn**> Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Xm**> Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias Conditions

Alias	Is preferred when
TST (shifted register)	Rd == '11111'

Operation

```
constant bits(datasize) operand1 = X[n, datasize];
constant bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount, datasize);

constant bits(datasize) result = operand1 AND operand2;
X[d, datasize] = result;
PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

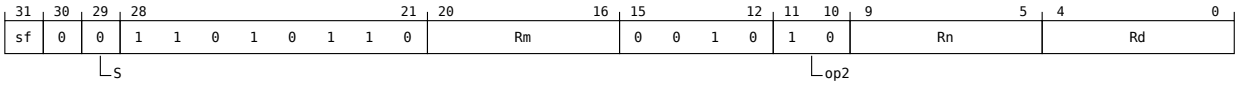
C6.2.15 ASR (register)

Arithmetic shift right (register)

This instruction shifts a register value right by a variable number of bits, shifting in copies of its sign bit, and writes the result to the destination register. The value of the second source register modulo the register size in bits gives the number of bits by which the first source register is right-shifted.

This is an alias of ASRV. This means:

- The encodings in this description are named to match the encodings of ASRV.
- The description of ASRV gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
ASR <Wd>, <Wn>, <Wm>
```

is equivalent to

```
ASRV <Wd>, <Wn>, <Wm>
```

and is always the preferred disassembly.

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
ASR <Xd>, <Xn>, <Xm>
```

is equivalent to

```
ASRV <Xd>, <Xn>, <Xm>
```

and is always the preferred disassembly.

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wn**> Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Wm**> Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the “Rm” field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Xn**> Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Xm**> Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the “Rm” field.

Operation

The description of ASRV gives the operational pseudocode for this instruction.

Operational Information

The description of ASRV gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

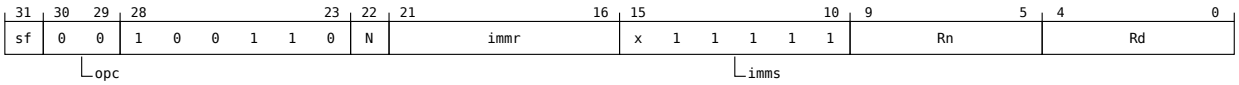
C6.2.16 ASR (immediate)

Arithmetic shift right (immediate)

This instruction shifts a register value right by an immediate number of bits, shifting in copies of the sign bit in the upper bits and zeros in the lower bits, and writes the result to the destination register.

This is an alias of SBFM. This means:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding for the 32-bit variant Applies when `(sf == 0 && N == 0 && imms == 011111)`

```
ASR <Wd>, <Wn>, #<shift>
```

is equivalent to

```
SBFM <Wd>, <Wn>, #<shift>, #31
```

and is always the preferred disassembly.

Encoding for the 64-bit variant Applies when `(sf == 1 && N == 1 && imms == 111111)`

```
ASR <Xd>, <Xn>, #<shift>
```

is equivalent to

```
SBFM <Xd>, <Xn>, #<shift>, #63
```

and is always the preferred disassembly.

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wn**> Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<**shift**> For the “32-bit” variant: is the shift amount, in the range 0 to 31, encoded in the “immr” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, encoded in the “immr” field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Xn**> Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Operational Information

The description of SBFM gives the operational information for this instruction.

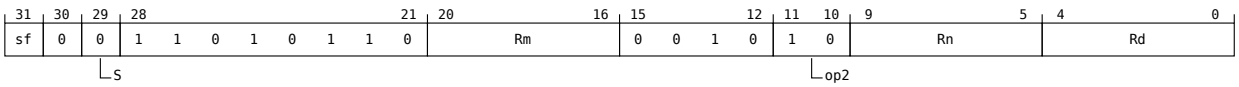
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.17 ASRV

Arithmetic shift right variable

This instruction shifts a register value right by a variable number of bits, shifting in copies of its sign bit, and writes the result to the destination register. The value of the second source register modulo the register size in bits gives the number of bits by which the first source register is right-shifted.

This instruction is used by the alias ASR (register).



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
ASRV <Wd>, <Wn>, <Wm>
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
ASRV <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
constant ShiftType shift_type = DecodeShift(op2);
```

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Wm> Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the “Rm” field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<Xm> Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the “Rm” field.

Operation

```
constant bits(datasize) operand2 = X[m, datasize];

X[d, datasize] = ShiftReg(n, shift_type, UInt(operand2) MOD datasize, datasize);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

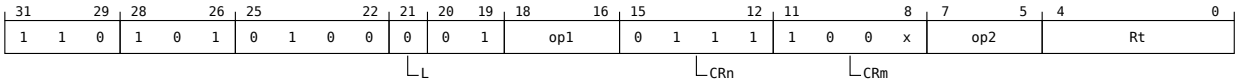
C6.2.18 AT

Address translate

For more information, see op0==0b01, cache maintenance, TLB maintenance, and address translation instructions.

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding

```
AT <at_op>, <Xt>
```

is equivalent to

```
SYS #<op1>, C7, <Cm>, #<op2>, <Xt>
```

and is the preferred disassembly when `SysOp(op1, '0111', CRm, op2) == Sys_AT`.

Assembler Symbols

<at_op> Is an AT operation name, as listed for the AT system instruction group, encoded in "op1:CRm:op2":

op1	CRm	op2	<at_op>	Architectural Feature
000	1000	000	S1E1R	-
000	1000	001	S1E1W	-
000	1000	010	S1E0R	-
000	1000	011	S1E0W	-
000	1001	000	S1E1RP	FEAT_PAN2
000	1001	001	S1E1WP	FEAT_PAN2
100	1000	000	S1E2R	-
100	1000	001	S1E2W	-
100	1000	100	S12E1R	-
100	1000	101	S12E1W	-
100	1000	110	S12E0R	-
100	1000	111	S12E0W	-
110	1000	000	S1E3R	-
110	1000	001	S1E3W	-

<Xt> Is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.

Operation

The description of SYS gives the operational pseudocode for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.19 AUTDA, AUTDZA

Authenticate data address, using key A

This instruction authenticates a data address, using a modifier and key A.

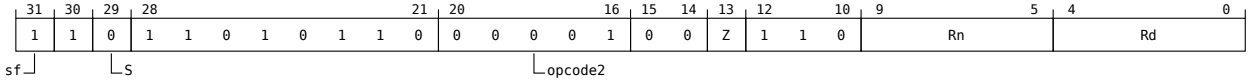
The address is in the general-purpose register that is specified by <Xd>.

The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xn|SP>, for AUTDA.
- The value zero, for AUTDZA.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. For information on behavior if the authentication fails, see Faulting on pointer authentication.

Integer
(FEAT_PAuth)



Encoding for the AUTDA variant Applies when (Z == 0)

```
AUTDA <Xd>, <Xn|SP>
```

Encoding for the AUTDZA variant Applies when (Z == 1 && Rn == 11111)

```
AUTDZA <Xd>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_UNDEF);
if Z == '1' && Rn != '11111' then EndOfDecode(Decode_UNDEF);
constant boolean auth_combined = FALSE;

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant boolean source_is_sp = Z == '0' && n == 31;
```

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn|SP> Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

Operation

```
if source_is_sp then
    X[d, 64] = AuthDA(X[d, 64], SP[64], auth_combined);
else
    X[d, 64] = AuthDA(X[d, 64], X[n, 64], auth_combined);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.20 AUTDB, AUTDZB

Authenticate data address, using key B

This instruction authenticates a data address, using a modifier and key B.

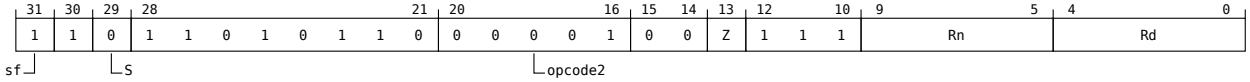
The address is in the general-purpose register that is specified by <Xd>.

The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xn|SP> for AUTDB.
- The value zero, for AUTDZB.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. For information on behavior if the authentication fails, see Faulting on pointer authentication.

Integer
(FEAT_PAuth)



Encoding for the AUTDB variant Applies when (Z == 0)

```
AUTDB <Xd>, <Xn|SP>
```

Encoding for the AUTDZB variant Applies when (Z == 1 && Rn == 11111)

```
AUTDZB <Xd>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_UNDEF);
if Z == '1' && Rn != '11111' then EndOfDecode(Decode_UNDEF);
constant boolean auth_combined = FALSE;

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant boolean source_is_sp = Z == '0' && n == 31;
```

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn|SP> Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

Operation

```
if source_is_sp then
    X[d, 64] = AuthDB(X[d, 64], SP[64], auth_combined);
else
    X[d, 64] = AuthDB(X[d, 64], X[n, 64], auth_combined);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.21 AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA

Authenticate instruction address, using key A

This instruction authenticates an instruction address, using a modifier and key A.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. For information on behavior if the authentication fails, see Faulting on pointer authentication.

The address is:

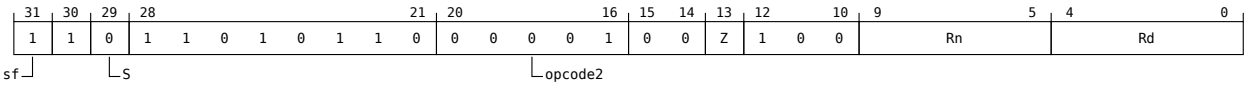
- In the general-purpose register that is specified by <Xd> for AUTIA and AUTIZA.
- In X17, for AUTIA1716.
- In X30, for AUTIASP and AUTIAZ.

The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xn|SP> for AUTIA.
- The value zero, for AUTIZA and AUTIAZ.
- In X16, for AUTIA1716.
- In SP, for AUTIASP.

It has encodings from 2 classes: Integer and System

Integer
(FEAT_PAuth)



Encoding for the AUTIA variant Applies when `(Z == 0)`

```
AUTIA <Xd>, <Xn|SP>
```

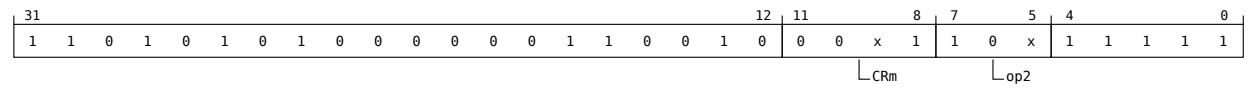
Encoding for the AUTIZA variant Applies when `(Z == 1 && Rn == 11111)`

```
AUTIZA <Xd>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_UNDEF);
if Z == '1' && Rn != '11111' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant boolean auth_combined = FALSE;

constant boolean source_is_sp = Z == '0' && n == 31;
```

System
(FEAT_PAuth)



Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the AUTIA1716 variant Applies when `(CRm == 0001 && op2 == 100)`

```
AUTIA1716
```

Encoding for the AUTIASP variant Applies when `(CRm == 0011 && op2 == 101)`

```
AUTIASP
```

Encoding for the AUTIAZ variant Applies when `(CRm == 0011 && op2 == 100)`

```
AUTIAZ
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_NOP);
integer d;
integer n;
boolean source_is_sp = FALSE;
constant boolean auth_combined = FALSE;

case CRm:op2 of
    when '0011 100' // AUTIAZ
        d = 30;
        n = 31;
    when '0011 101' // AUTIASP
        d = 30;
        source_is_sp = TRUE;
    when '0001 100' // AUTIA1716
        d = 17;
        n = 16;
```

Assembler Symbols

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Xn**|**SP**> Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the “Rn” field.

Operation

```
if source_is_sp then
    X[d, 64] = AuthIA(X[d, 64], SP[64], auth_combined);
else
    X[d, 64] = AuthIA(X[d, 64], X[n, 64], auth_combined);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.22 AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB

Authenticate instruction address, using key B

This instruction authenticates an instruction address, using a modifier and key B.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. For information on behavior if the authentication fails, see Faulting on pointer authentication.

The address is:

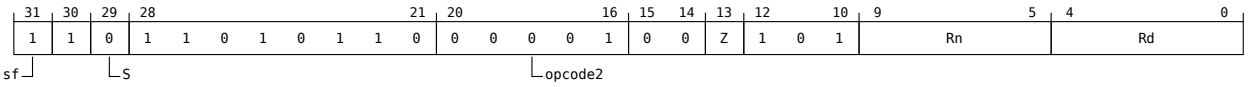
- In the general-purpose register that is specified by <Xd> for AUTIB and AUTIZB.
- In X17, for AUTIB1716.
- In X30, for AUTIBSP and AUTIBZ.

The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xn|SP> for AUTIB.
- The value zero, for AUTIZB and AUTIBZ.
- In X16, for AUTIB1716.
- In SP, for AUTIBSP.

It has encodings from 2 classes: Integer and System

Integer
(FEAT_PAuth)



Encoding for the AUTIB variant Applies when (Z == 0)

```
AUTIB <Xd>, <Xn|SP>
```

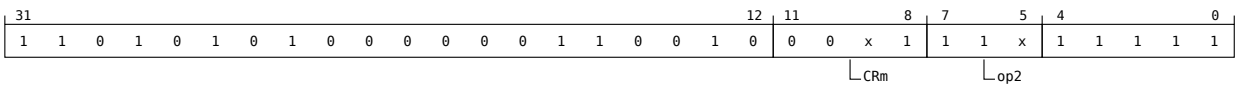
Encoding for the AUTIZB variant Applies when (Z == 1 && Rn == 11111)

```
AUTIZB <Xd>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_UNDEF);
if Z == '1' && Rn != '11111' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant boolean auth_combined = FALSE;

constant boolean source_is_sp = Z == '0' && n == 31;
```

System
(FEAT_PAuth)



Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the AUTIB1716 variant Applies when `(CRm == 0001 && op2 == 110)`

```
AUTIB1716
```

Encoding for the AUTIBSP variant Applies when `(CRm == 0011 && op2 == 111)`

```
AUTIBSP
```

Encoding for the AUTIBZ variant Applies when `(CRm == 0011 && op2 == 110)`

```
AUTIBZ
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_NOP);
integer d;
integer n;
boolean source_is_sp = FALSE;
constant boolean auth_combined = FALSE;

case CRm:op2 of
    when '0011 110' // AUTIBZ
        d = 30;
        n = 31;
    when '0011 111' // AUTIBSP
        d = 30;
        source_is_sp = TRUE;
    when '0001 110' // AUTIB1716
        d = 17;
        n = 16;
```

Assembler Symbols

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Xn**|**SP**> Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

Operation

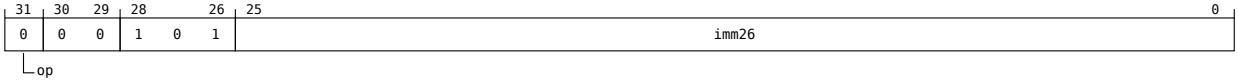
```
if source_is_sp then
    X[d, 64] = AuthIB(X[d, 64], SP[64], auth_combined);
else
    X[d, 64] = AuthIB(X[d, 64], X[n, 64], auth_combined);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.23 B

Branch

This instruction branches unconditionally to a label at a PC-relative offset. This instruction provides a hint that this is not a subroutine call or return.



Encoding

```
B <label>
```

Decode for this encoding

```
constant bits(64) offset = SignExtend(imm26:'00', 64);
```

Assembler Symbols

<label> Is the program label to be unconditionally branched to. Its offset from the address of this instruction, in the range +/-128MB, is encoded as “imm26” times 4.

Operation

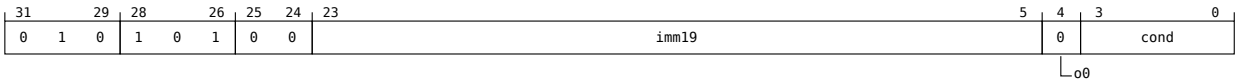
```
constant boolean branch_conditional = FALSE;
BranchTo(PC64 + offset, BranchType_DIR, branch_conditional);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.24 B.cond

Branch conditionally

This instruction branches conditionally to a label at a PC-relative offset. This instruction provides a hint that this is not a subroutine call or return.



Encoding

```
B.<cond> <label>
```

Decode for this encoding

```
constant bits(64) offset = SignExtend(imm19:'00', 64);
constant bits(4) condition = cond;
```

Assembler Symbols

<**cond**> Is one of the standard conditions, encoded in the standard way, and encoded in "cond":

cond	<cond>
0000	EQ
0001	NE
0010	CS
0011	CC
0100	MI
0101	PL
0110	VS
0111	VC
1000	HI
1001	LS
1010	GE
1011	LT
1100	GT
1101	LE
1110	AL
1111	NV

<**label**> Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

Operation

```
constant boolean branch_conditional = TRUE;
if ConditionHolds(condition) then
    BranchTo(PC64 + offset, BranchType_DIR, branch_conditional);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.25 BFC

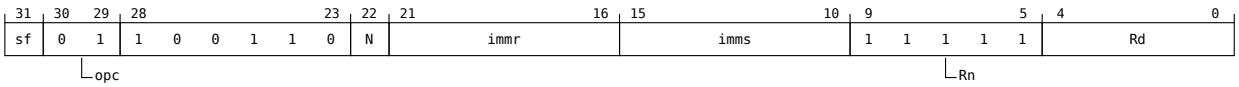
Bitfield clear

This instruction sets a bitfield of <width> bits at bit position <lsb> of the destination register to zero, leaving the other destination bits unchanged.

This is an alias of BFM. This means:

- The encodings in this description are named to match the encodings of BFM.
- The description of BFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Leaving other bits unchanged
(FEAT_ASMv8p2)



Encoding for the 32-bit variant Applies when `(sf == 0 && N == 0)`

```
BFC <Wd>, #<lsb>, #<width>
```

is equivalent to

```
BFM <Wd>, WZR, #(-<lsb> MOD 32), #(<width>-1)
```

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

Encoding for the 64-bit variant Applies when `(sf == 1 && N == 1)`

```
BFC <Xd>, #<lsb>, #<width>
```

is equivalent to

```
BFM <Xd>, XZR, #(-<lsb> MOD 64), #(<width>-1)
```

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**lsb**> For the "32-bit" variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 31.

For the "64-bit" variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 63.

<**width**> For the "32-bit" variant: is the width of the bitfield, in the range 1 to 32-<lsb>.

For the "64-bit" variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

Operation

The description of BFM gives the operational pseudocode for this instruction.

Operational Information

The description of BFM gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

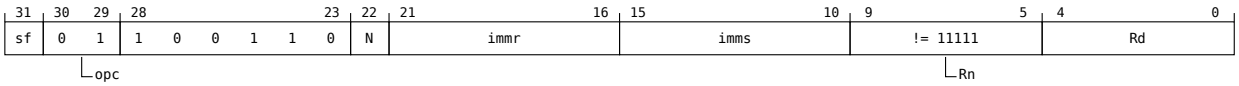
C6.2.26 BFI

Bitfield insert

This instruction copies a bitfield of <width> bits from the least significant bits of the source register to bit position <lsb> of the destination register, leaving the other destination bits unchanged.

This is an alias of BFM. This means:

- The encodings in this description are named to match the encodings of BFM.
- The description of BFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding for the 32-bit variant Applies when `(sf == 0 && N == 0)`

```
BFI <Wd>, <Wn>, #<lsb>, #<width>
```

is equivalent to

```
BFM <Wd>, <Wn>, #(-<lsb> MOD 32), #(<width>-1)
```

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

Encoding for the 64-bit variant Applies when `(sf == 1 && N == 1)`

```
BFI <Xd>, <Xn>, #<lsb>, #<width>
```

is equivalent to

```
BFM <Xd>, <Xn>, #(-<lsb> MOD 64), #(<width>-1)
```

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Wn**> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

<**lsb**> For the "32-bit" variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 31.

For the "64-bit" variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 63.

<**width**> For the "32-bit" variant: is the width of the bitfield, in the range 1 to 32-<lsb>.

For the "64-bit" variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Xn**> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of BFM gives the operational pseudocode for this instruction.

Operational Information

The description of BFM gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.27 BFM

Bitfield move

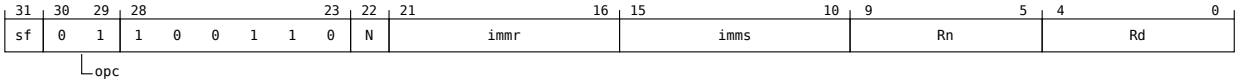
This instruction is usually accessed via one of its aliases, which are always preferred for disassembly.

If <imms> is greater than or equal to <immr>, this copies a bitfield of (<imms>-<immr>+1) bits starting from bit position <immr> in the source register to the least significant bits of the destination register.

If <imms> is less than <immr>, this copies a bitfield of (<imms>+1) bits from the least significant bits of the source register to bit position (regsize-<immr>) of the destination register, where regsize is the destination register size of 32 or 64 bits.

In both cases, the other bits of the destination register remain unchanged.

This instruction is used by the aliases BFC, BFI, and BFXIL.



Encoding for the 32-bit variant Applies when `(sf == 0 && N == 0)`

```
BFM <Wd>, <Wn>, #<immr>, #<imms>
```

Encoding for the 64-bit variant Applies when `(sf == 1 && N == 1)`

```
BFM <Xd>, <Xn>, #<immr>, #<imms>
```

Decode for all variants of this encoding

```
if sf == '1' && N != '1' then EndOfDecode(Decode_UNDEF);
if sf == '0' && (N != '0' || immr<5> != '0' || imms<5> != '0') then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);
constant integer s = UInt(imms);
constant integer r = UInt(immr);

bits(datasize) wmask;
bits(datasize) tmask;
(wmask, tmask) = DecodeBitMasks(N, imms, immr, FALSE, datasize);
```

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Wn**> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

<**immr**> For the "32-bit" variant: is the right rotate amount, in the range 0 to 31, encoded in the "immr" field.

For the "64-bit" variant: is the right rotate amount, in the range 0 to 63, encoded in the "immr" field.

<**imms**> For the "32-bit" variant: is the leftmost bit number to be moved from the source, in the range 0 to 31, encoded in the "imms" field.

For the "64-bit" variant: is the leftmost bit number to be moved from the source, in the range 0 to 63, encoded in the "imms" field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**Xn**> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
BFC	`Rn == '11111' && UInt(imms) < UInt(immr)`
BFI	`Rn != '11111' && UInt(imms) < UInt(immr)`
BFXIL	`UInt(imms) >= UInt(immr)`

Operation

```
constant bits(datasize) dst = X[d, datasize];
constant bits(datasize) src = X[n, datasize];

// Perform bitfield move on low bits
constant bits(datasize) bot = (dst AND NOT(wmask)) OR (ROR(src, r) AND wmask);

// Combine extension bits and result bits
X[d, datasize] = (dst AND NOT(tmask)) OR (bot AND tmask);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

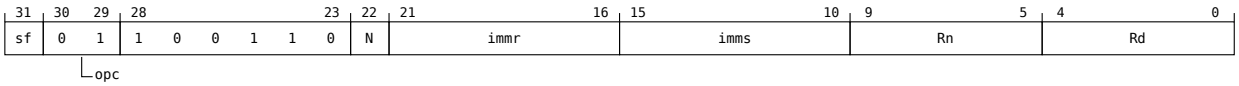
C6.2.28 BFXIL

Bitfield extract and insert at low end

This instruction copies a bitfield of <width> bits starting from bit position <lsb> in the source register to the least significant bits of the destination register, leaving the other destination bits unchanged.

This is an alias of BFM. This means:

- The encodings in this description are named to match the encodings of BFM.
- The description of BFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding for the 32-bit variant Applies when `(sf == 0 && N == 0)`

```
BFXIL <Wd>, <Wn>, #<lsb>, #<width>
```

is equivalent to

```
BFM <Wd>, <Wn>, #<lsb>, #(<lsb>+<width>-1)
```

and is the preferred disassembly when `UInt(imms) >= UInt(immr)`.

Encoding for the 64-bit variant Applies when `(sf == 1 && N == 1)`

```
BFXIL <Xd>, <Xn>, #<lsb>, #<width>
```

is equivalent to

```
BFM <Xd>, <Xn>, #<lsb>, #(<lsb>+<width>-1)
```

and is the preferred disassembly when `UInt(imms) >= UInt(immr)`.

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Wn**> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

<**lsb**> For the "32-bit" variant: is the bit number of the lsb of the source bitfield, in the range 0 to 31.

For the "64-bit" variant: is the bit number of the lsb of the source bitfield, in the range 0 to 63.

<**width**> For the "32-bit" variant: is the width of the bitfield, in the range 1 to 32-<lsb>.

For the "64-bit" variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Xn**> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of BFM gives the operational pseudocode for this instruction.

Operational Information

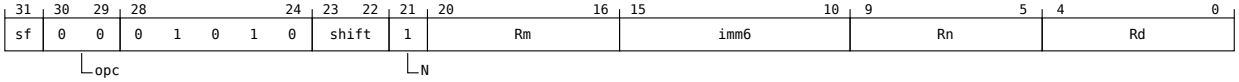
The description of BFM gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.29 BIC (shifted register)

Bitwise bit clear (shifted register)

This instruction performs a bitwise AND of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
BIC <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
BIC <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

Decode for all variants of this encoding

```
if sf == '0' && imm6<5> == '1' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
constant ShiftType shift_type = DecodeShift(shift);
constant integer shift_amount = UInt(imm6);
```

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wn**> Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Wm**> Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<**shift**> Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:

shift	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<**amount**> For the “32-bit” variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm6” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Xn**> Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Xm**> Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
constant bits(datasize) operand1 = X[n, datasize];
constant bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount, datasize);

X[d, datasize] = operand1 AND NOT(operand2);
```

Operational information

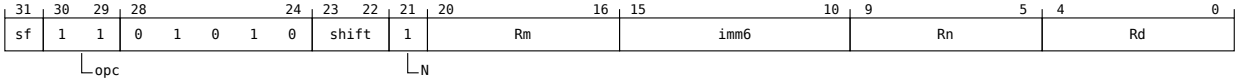
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.30 BICS (shifted register)

Bitwise bit clear (shifted register), setting flags

This instruction performs a bitwise AND of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
BICS <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
BICS <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

Decode for all variants of this encoding

```
if sf == '0' && imm6<5> == '1' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
constant ShiftType shift_type = DecodeShift(shift);
constant integer shift_amount = UInt(imm6);
```

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wn**> Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Wm**> Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<**shift**> Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:

shift	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<**amount**> For the “32-bit” variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm6” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Xn**> Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Xm**> Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
constant bits(datasize) operand1 = X[n, datasize];
constant bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount, datasize);

constant bits(datasize) result = operand1 AND NOT(operand2);
X[d, datasize] = result;
PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';
```

Operational information

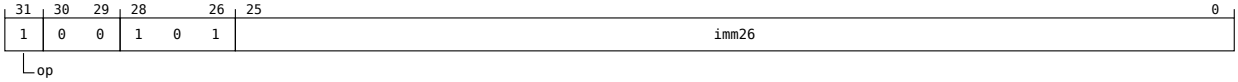
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.31 BL

Branch with link

This instruction branches to a PC-relative offset, setting register X30 to PC+4. This instruction provides a hint that this is a subroutine call.



Encoding

```
BL <label>
```

Decode for this encoding

```
constant bits(64) offset = SignExtend(imm26:'00', 64);
constant integer d = 30;
```

Assembler Symbols

<label> Is the program label to be unconditionally branched to. Its offset from the address of this instruction, in the range +/-128MB, is encoded as "imm26" times 4.

Operation

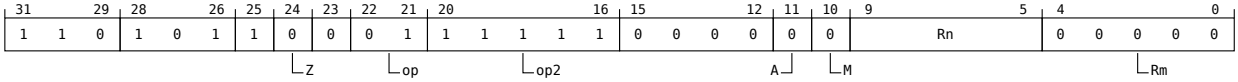
```
X[d, 64] = PC64 + 4;

constant boolean branch_conditional = FALSE;
BranchTo(PC64 + offset, BranchType_DIRCALL, branch_conditional);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.32 BLR

Branch with link to register

This instruction calls a subroutine at an address in a register, setting register X30 to PC+4. This instruction provides a hint that this is a subroutine call.



Encoding

BLR <Xn>

Decode for this encoding

```
constant integer n = UInt(Rn);
```

Assembler Symbols

<Xn> Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field.

Operation

```
constant bits(64) target = X[n, 64];

X[30, 64] = PC64 + 4;

constant boolean branch_conditional = FALSE;
BranchTo(target, BranchType_INDCALL, branch_conditional);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.33 BLRAA, BLRAAZ, BLRAB, BLRABZ

Branch with link to register, with pointer authentication

This instruction authenticates the address in the general-purpose register that is specified by <Xn>, using a modifier and the specified key, and calls a subroutine at the authenticated address, setting register X30 to PC+4.

The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xm|SP>, for BLRAA and BLRAB.
- The value zero, for BLRAAZ and BLRABZ.

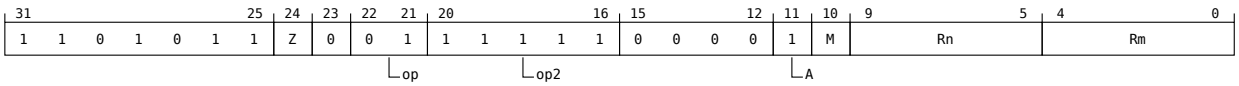
Key A is used for BLRAA and BLRAAZ. Key B is used for BLRAB and BLRABZ.

If the authentication passes, the PE continues execution at the target of the branch. For information on behavior if the authentication fails, see Faulting on pointer authentication.

The authenticated address is not written back to the general-purpose register.

This instruction provides a hint that this is a subroutine call.

Integer
(FEAT_PAuth)



Encoding for the Key A, register modifier variant Applies when `(Z == 1 && M == 0)`

```
BLRAA <Xn>, <Xm|SP>
```

Encoding for the Key A, zero modifier variant Applies when `(Z == 0 && M == 0 && Rm == 11111)`

```
BLRAAZ <Xn>
```

Encoding for the Key B, register modifier variant Applies when `(Z == 1 && M == 1)`

```
BLRAB <Xn>, <Xm|SP>
```

Encoding for the Key B, zero modifier variant Applies when `(Z == 0 && M == 1 && Rm == 11111)`

```
BLRABZ <Xn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_UNDEF);
if Z == '0' && Rm != '11111' then EndOfDecode(Decode_UNDEF);

constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant boolean use_key_a = (M == '0');
constant boolean source_is_sp = ((Z == '1') && (m == 31));
constant boolean auth_then_branch = TRUE;
```

Assembler Symbols

<Xn> Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xm|SP> Is the 64-bit name of the general-purpose source register or stack pointer holding the modifier, encoded in the "Rm" field.

Operation

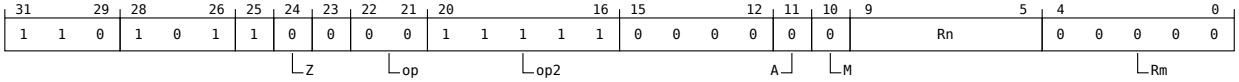
```
bits(64) target = X[n, 64];
constant bits(64) modifier = if source_is_sp then SP[64] else X[m, 64];

if use_key_a then
    target = AuthIA(target, modifier, auth_then_branch);
else
    target = AuthIB(target, modifier, auth_then_branch);

X[30, 64] = PC64 + 4;

constant boolean branch_conditional = FALSE;
BranchTo(target, BranchType_INDCALL, branch_conditional);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.34 BR

Branch to register

This instruction branches unconditionally to an address in a register. This instruction provides a hint that this is not a subroutine call or return.



Encoding

```
BR <Xn>
```

Decode for this encoding

```
constant integer n = UInt(Rn);
```

Assembler Symbols

<**Xn**> Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field.

Operation

```
constant bits(64) target = X[n, 64];

constant boolean branch_conditional = FALSE;
BranchTo(target, BranchType_INDIR, branch_conditional);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.35 BRAA, BRAAZ, BRAB, BRABZ

Branch to register, with pointer authentication

This instruction authenticates the address in the general-purpose register that is specified by <Xn>, using a modifier and the specified key, and branches to the authenticated address.

The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xm|SP>, for BRAA and BRAB.
- The value zero, for BRAAZ and BRABZ.

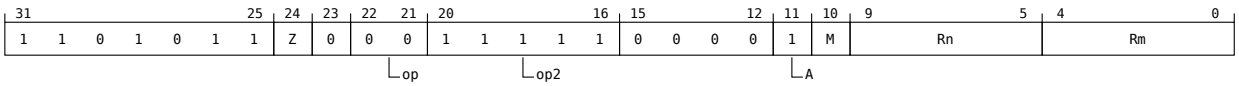
Key A is used for BRAA and BRAAZ. Key B is used for BRAB and BRABZ.

If the authentication passes, the PE continues execution at the target of the branch. For information on behavior if the authentication fails, see Faulting on pointer authentication.

The authenticated address is not written back to the general-purpose register.

This instruction provides a hint that this is not a subroutine call or return.

Integer
(FEAT_PAuth)



Encoding for the Key A, register modifier variant Applies when (Z == 1 && M == 0)

```
BRAA <Xn>, <Xm|SP>
```

Encoding for the Key A, zero modifier variant Applies when (Z == 0 && M == 0 && Rm == 11111)

```
BRAAZ <Xn>
```

Encoding for the Key B, register modifier variant Applies when (Z == 1 && M == 1)

```
BRAB <Xn>, <Xm|SP>
```

Encoding for the Key B, zero modifier variant Applies when (Z == 0 && M == 1 && Rm == 11111)

```
BRABZ <Xn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_UNDEF);
if Z == '0' && Rm != '11111' then EndOfDecode(Decode_UNDEF);

constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant boolean use_key_a = (M == '0');
constant boolean source_is_sp = ((Z == '1') && (m == 31));
constant boolean auth_then_branch = TRUE;
```

Assembler Symbols

<**Xn**> Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xm|SP> Is the 64-bit name of the general-purpose source register or stack pointer holding the modifier, encoded in the "Rm" field.

Operation

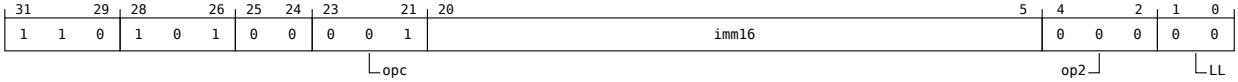
```
bits(64) target = X[n, 64];
constant bits(64) modifier = if source_is_sp then SP[64] else X[m, 64];

if use_key_a then
    target = AuthIA(target, modifier, auth_then_branch);
else
    target = AuthIB(target, modifier, auth_then_branch);

constant boolean branch_conditional = FALSE;
BranchTo(target, BranchType_INDIR, branch_conditional);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.36 BRK

Breakpoint instruction

This instruction generates a Breakpoint Instruction exception. The PE records the exception in ESR_ELx, using the EC value 0x3C, and captures the value of the immediate argument in ESR_ELx.ISS.



Encoding

```
BRK #<imm>
```

Decode for this encoding

```
constant bits(16) comment = imm16;
```

Assembler Symbols

<**imm**> Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the “imm16” field.

Operation

```
AArch64.SoftwareBreakpoint(comment);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.37 CAS, CASA, CASAL, CASL

Compare and swap word or doubleword in memory

This instruction reads a 32-bit word or 64-bit doubleword from memory, and compares it against the value held in a first register. If the comparison is equal, the value in a second register is written to memory. If the comparison is not equal, the architecture permits writing the value read from the location to memory. If the write is performed, the read and write occur atomically such that no other modification of the memory location can take place between the read and write.

- If the destination register is not one of WZR or XZR, CASA and CASAL load from memory with acquire semantics.
- CASL and CASAL store to memory with release semantics.
- CAS has neither acquire nor release semantics.

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails.

If the instruction generates a synchronous Data Abort, the register which is compared and loaded, that is <Ws>, or <Xs>, is restored to the value held in the register before the instruction was executed.

For a CAS or CASA instruction, when <Ws> or <Xs> specifies the same register as <Wt> or <Xt>, this signals to the memory system that an additional subsequent CAS, CASA, CASAL, or CASL access to the specified location is expected to occur in the near future in the absence of taking an exception or exceptional software conditions. The memory system can respond by taking actions that are expected to enable the subsequent CAS, CASA, CASAL, or CASL access to succeed when it does occur.

A code sequence starting with a CAS or CASA instruction for which <Ws> or <Xs> specifies the same register as <Wt> or <Xt>, and ending with a subsequent CAS, CASA, CASAL, or CASL to the same location, exhibits the following properties for best performance when the location may be accessed concurrently, on one or more other PEs:

- The sequence does not contain any direct system register writes, address translation instructions, cache or TLB maintenance operations, exception producing instructions, exception returns, or ISB barriers.
- The execution of the sequence includes 32 or fewer instructions.
- The value provided in <Ws> or <Xs> of the first CAS or CASA is a value likely to result in the comparison failing. A failing comparison result may lead to better performance due to the hardware not performing a write to memory.

Note

For a CAS or CASA instruction, when <Ws> or <Xs> specifies the same register as <Wt> or <Xt>, the value in memory is not modified, because the CAS or CASA either fails its compare or writes the same value back to memory.

For performance reasons, software is advised to not use a CAS or CASA where <Ws> or <Xs> specify the same register as <Wt> or <Xt> in a routine which includes a subsequent CAS, CASA, CASAL, or CASL for any of the following purposes:

- To load the value that will be compared to conditionally determine whether or not to execute the subsequent CAS, CASA, CASAL, or CASL.
- As a form of prefetch executed between the instruction loading the value that will be compared and the subsequent compare instruction which conditionally determines whether or not to execute the subsequent CAS, CASA, CASAL, or CASL.

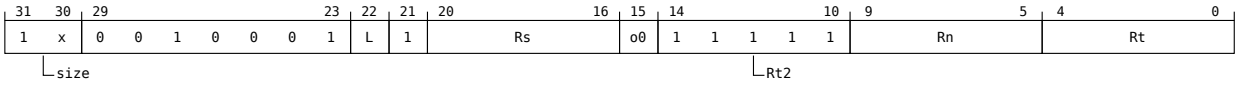
This guidance does not apply in cases where the compare between the CAS or CASA, where <Ws> or <Xs> specify the same register as <Wt> or <Xt>, and the subsequent CAS, CASA, CASAL, or CASL is intended to detect an exceptional software condition such that the result of the compare does not prevent the subsequent CAS, CASA, CASAL, or CASL from executing under normal operating conditions.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

No offset
(FEAT_LSE)

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential



Encoding for the 32-bit CAS variant Applies when `(size == 10 && L == 0 && o0 == 0)`

```
CAS <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the 32-bit CASA variant Applies when `(size == 10 && L == 1 && o0 == 0)`

```
CASA <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the 32-bit CASAL variant Applies when `(size == 10 && L == 1 && o0 == 1)`

```
CASAL <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the 32-bit CASL variant Applies when `(size == 10 && L == 0 && o0 == 1)`

```
CASL <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit CAS variant Applies when `(size == 11 && L == 0 && o0 == 0)`

```
CAS <Xs>, <Xt>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit CASA variant Applies when `(size == 11 && L == 1 && o0 == 0)`

```
CASA <Xs>, <Xt>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit CASAL variant Applies when `(size == 11 && L == 1 && o0 == 1)`

```
CASAL <Xs>, <Xt>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit CASL variant Applies when `(size == 11 && L == 0 && o0 == 1)`

```
CASL <Xs>, <Xt>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer datasize = 8 << UInt(size);
constant integer regsize = if datasize == 64 then 64 else 32;
constant boolean acquire = L == '1' && t != 31;
constant boolean release = o0 == '1';
```

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register to be compared and loaded, encoded in the “Rs” field.

<Wt> Is the 32-bit name of the general-purpose register to be conditionally stored, encoded in the “Rt” field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs> Is the 64-bit name of the general-purpose register to be compared and loaded, encoded in the “Rs” field.

<Xt> Is the 64-bit name of the general-purpose register to be conditionally stored, encoded in the “Rt” field.

Operation

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
bits(64) address;
bits(datasize) comparevalue;
bits(datasize) newvalue;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_CAS, acquire, release,
                                                          privileged, t, s);
comparevalue = X[s, datasize];
newvalue = X[t, datasize];

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(datasize) data = MemAtomic(address, comparevalue, newvalue, accdesc);
X[s, regsize] = ZeroExtend(data, regsize);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.38 CASB, CASAB, CASALB, CASLB

Compare and swap byte in memory

This instruction reads an 8-bit byte from memory, and compares it against the value held in a first register. If the comparison is equal, the value in a second register is written to memory. If the comparison is not equal, the architecture permits writing the value read from the location to memory. If the write is performed, the read and write occur atomically such that no other modification of the memory location can take place between the read and write.

- If the destination register is not one of WZR or XZR, CASAB and CASALB load from memory with acquire semantics.
- CASLB and CASALB store to memory with release semantics.
- CASB has neither acquire nor release semantics.

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails.

If the instruction generates a synchronous Data Abort, the register which is compared and loaded, that is <Ws>, is restored to the values held in the register before the instruction was executed.

For a CASB or CASAB instruction, when <Ws> or <Xs> specifies the same register as <Wt> or <Xt>, this signals to the memory system that an additional subsequent CASB, CASAB, CASALB, or CASLB access to the specified location is expected to occur in the near future in the absence of taking an exception or exceptional software conditions. The memory system can respond by taking actions that are expected to enable the subsequent CASB, CASAB, CASALB, or CASLB access to succeed when it does occur.

A code sequence starting with a CASB or CASAB instruction for which <Ws> or <Xs> specifies the same register as <Wt> or <Xt>, and ending with a subsequent CASB, CASAB, CASALB, or CASLB to the same location, exhibits the following properties for best performance when the location may be accessed concurrently, on one or more other PEs:

- The sequence does not contain any direct system register writes, address translation instructions, cache or TLB maintenance operations, exception producing instructions, exception returns, or ISB barriers.
- The execution of the sequence includes 32 or fewer instructions.
- The value provided in <Ws> or <Xs> of the first CASB or CASAB is a value likely to result in the comparison failing. A failing comparison result may lead to better performance due to the hardware not performing a write to memory.

Note

For a CASB or CASAB instruction, when <Ws> or <Xs> specifies the same register as <Wt> or <Xt>, the value in memory is not modified, because the CASB or CASAB either fails its compare or writes the same value back to memory.

For performance reasons, software is advised to not use a CASB or CASAB where <Ws> or <Xs> specify the same register as <Wt> or <Xt> in a routine which includes a subsequent CASB, CASAB, CASALB, or CASLB for any of the following purposes:

- To load the value that will be compared to conditionally determine whether or not to execute the subsequent CASB, CASAB, CASALB, or CASLB.
- As a form of prefetch executed between the instruction loading the value that will be compared and the subsequent compare instruction which conditionally determines whether or not to execute the subsequent CASB, CASAB, CASALB, or CASLB.

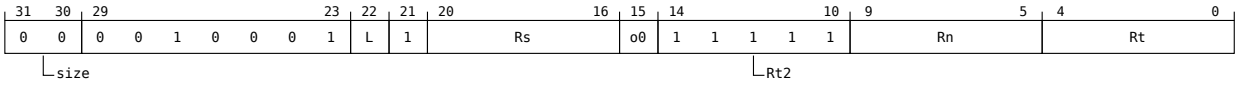
This guidance does not apply in cases where the compare between the CASB or CASAB, where <Ws> or <Xs> specify the same register as <Wt> or <Xt>, and the subsequent CASB, CASAB, CASALB, or CASLB is intended to detect an exceptional software condition such that the result of the compare does not prevent the subsequent CASB, CASAB, CASALB, or CASLB from executing under normal operating conditions.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

No offset
(FEAT_LSE)

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential



Encoding for the CASB variant Applies when (L == 0 && o0 == 0)

```
CASB <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the CASAB variant Applies when (L == 1 && o0 == 0)

```
CASAB <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the CASALB variant Applies when (L == 1 && o0 == 1)

```
CASALB <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the CASLB variant Applies when (L == 0 && o0 == 1)

```
CASLB <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant boolean acquire = L == '1' && t != 31;
constant boolean release = o0 == '1';
```

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register to be compared and loaded, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be conditionally stored, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) comparevalue;
bits(8) newvalue;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_CAS, acquire, release,
                                                          privileged, t, s);
comparevalue = X[s, 8];
newvalue = X[t, 8];

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(8) data = MemAtomic(address, comparevalue, newvalue, accdesc);
X[s, 32] = ZeroExtend(data, 32);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.39 CASH, CASAH, CASALH, CASLH

Compare and swap halfword in memory

This instruction reads a 16-bit halfword from memory, and compares it against the value held in a first register. If the comparison is equal, the value in a second register is written to memory. If the comparison is not equal, the architecture permits writing the value read from the location to memory. If the write is performed, the read and write occur atomically such that no other modification of the memory location can take place between the read and write.

- If the destination register is not one of WZR or XZR, CASAH and CASALH load from memory with acquire semantics.
- CASLH and CASALH store to memory with release semantics.
- CASH has neither acquire nor release semantics.

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails.

If the instruction generates a synchronous Data Abort, the register which is compared and loaded, that is <Ws>, is restored to the values held in the register before the instruction was executed.

For a CASH or CASAH instruction, when <Ws> or <Xs> specifies the same register as <Wt> or <Xt>, this signals to the memory system that an additional subsequent CASH, CASAH, CASALH, or CASLH access to the specified location is expected to occur in the near future in the absence of taking an exception or exceptional software conditions. The memory system can respond by taking actions that are expected to enable the subsequent CASH, CASAH, CASALH, or CASLH access to succeed when it does occur.

A code sequence starting with a CASH or CASAH instruction for which <Ws> or <Xs> specifies the same register as <Wt> or <Xt>, and ending with a subsequent CASH, CASAH, CASALH, or CASLH to the same location, exhibits the following properties for best performance when the location may be accessed concurrently, on one or more other PEs:

- The sequence does not contain any direct system register writes, address translation instructions, cache or TLB maintenance operations, exception producing instructions, exception returns, or ISB barriers.
- The execution of the sequence includes 32 or fewer instructions.
- The value provided in <Ws> or <Xs> of the first CASH or CASAH is a value likely to result in the comparison failing. A failing comparison result may lead to better performance due to the hardware not performing a write to memory.

Note

For a CASH or CASAH instruction, when <Ws> or <Xs> specifies the same register as <Wt> or <Xt>, the value in memory is not modified, because the CASH or CASAH either fails its compare or writes the same value back to memory.

For performance reasons, software is advised to not use a CASH or CASAH where <Ws> or <Xs> specify the same register as <Wt> or <Xt> in a routine which includes a subsequent CASH, CASAH, CASALH, or CASLH for any of the following purposes:

- To load the value that will be compared to conditionally determine whether or not to execute the subsequent CASH, CASAH, CASALH, or CASLH.
- As a form of prefetch executed between the instruction loading the value that will be compared and the subsequent compare instruction which conditionally determines whether or not to execute the subsequent CASH, CASAH, CASALH, or CASLH.

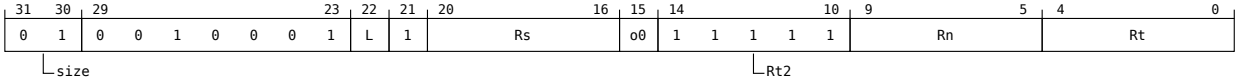
This guidance does not apply in cases where the compare between the CASH or CASAH, where <Ws> or <Xs> specify the same register as <Wt> or <Xt>, and the subsequent CASH, CASAH, CASALH, or CASLH is intended to detect an exceptional software condition such that the result of the compare does not prevent the subsequent CASH, CASAH, CASALH, or CASLH from executing under normal operating conditions.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

No offset
(FEAT_LSE)

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential



Encoding for the CASH variant Applies when `(L == 0 && o0 == 0)`

```
CASH <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the CASAH variant Applies when `(L == 1 && o0 == 0)`

```
CASAH <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the CASALH variant Applies when `(L == 1 && o0 == 1)`

```
CASALH <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the CASLH variant Applies when `(L == 0 && o0 == 1)`

```
CASLH <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant boolean acquire = L == '1' && t != 31;
constant boolean release = o0 == '1';
```

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register to be compared and loaded, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be conditionally stored, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) comparevalue;
bits(16) newvalue;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_CAS, acquire, release,
                                                          privileged, t, s);
comparevalue = X[s, 16];
newvalue = X[t, 16];

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(16) data = MemAtomic(address, comparevalue, newvalue, accdesc);
X[s, 32] = ZeroExtend(data, 32);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.40 CASP, CASPA, CASPAL, CASPL

Compare and swap pair of words or doublewords in memory

This instruction reads a pair of 32-bit words or 64-bit doublewords from memory, and compares them against the values held in the first pair of registers. If the comparison is equal, the values in the second pair of registers are written to memory. If the comparison is not equal, the architecture permits writing the value read from the location to memory. If the writes are performed, the reads and writes occur atomically such that no other modification of the memory location can take place between the reads and writes.

- CASPA and CASPAL load from memory with acquire semantics.
- CASPL and CASPAL store to memory with release semantics.
- CASP has neither acquire nor release semantics.

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails.

If the instruction generates a synchronous Data Abort, the registers which are compared and loaded, that is <Ws> and <W(s+1)>, or <Xs> and <X(s+1)>, are restored to the values held in the registers before the instruction was executed.

For a CASP or CASPA instruction, when <Ws> or <Xs> specifies the same register as <Wt> or <Xt>, this signals to the memory system that an additional subsequent CASP, CASPA, CASPAL, or CASPL access to the specified location is expected to occur in the near future in the absence of taking an exception or exceptional software conditions. The memory system can respond by taking actions that are expected to enable the subsequent CASP, CASPA, CASPAL, or CASPL access to succeed when it does occur.

A code sequence starting with a CASP or CASPA instruction for which <Ws> or <Xs> specifies the same register as <Wt> or <Xt>, and ending with a subsequent CASP, CASPA, CASPAL, or CASPL to the same location, exhibits the following properties for best performance when the location may be accessed concurrently, on one or more other PEs:

- The sequence does not contain any direct system register writes, address translation instructions, cache or TLB maintenance operations, exception producing instructions, exception returns, or ISB barriers.
- The execution of the sequence includes 32 or fewer instructions.
- The value provided in <Ws> or <Xs> of the first CASP or CASPA is a value likely to result in the comparison failing. A failing comparison result may lead to better performance due to the hardware not performing a write to memory.

Note

For a CASP or CASPA instruction, when <Ws> or <Xs> specifies the same register as <Wt> or <Xt>, the value in memory is not modified, because the CASP or CASPA either fails its compare or writes the same value back to memory.

For performance reasons, software is advised to not use a CASP or CASPA where <Ws> or <Xs> specify the same register as <Wt> or <Xt> in a routine which includes a subsequent CASP, CASPA, CASPAL, or CASPL for any of the following purposes:

- To load the value that will be compared to conditionally determine whether or not to execute the subsequent CASP, CASPA, CASPAL, or CASPL.
- As a form of prefetch executed between the instruction loading the value that will be compared and the subsequent compare instruction which conditionally determines whether or not to execute the subsequent CASP, CASPA, CASPAL, or CASPL.

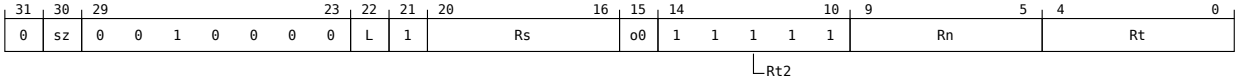
This guidance does not apply in cases where the compare between the CASP or CASPA, where <Ws> or <Xs> specify the same register as <Wt> or <Xt>, and the subsequent CASP, CASPA, CASPAL, or CASPL is intended to detect an exceptional software condition such that the result of the compare does not prevent the subsequent CASP, CASPA, CASPAL, or CASPL from executing under normal operating conditions.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

No offset
(FEAT_LSE)



Encoding for the 32-bit CASP variant Applies when `(sz == 0 && L == 0 && o0 == 0)`

```
CASP <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Xn|SP>{, #0}]
```

Encoding for the 32-bit CASPA variant Applies when `(sz == 0 && L == 1 && o0 == 0)`

```
CASPA <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Xn|SP>{, #0}]
```

Encoding for the 32-bit CASPAL variant Applies when `(sz == 0 && L == 1 && o0 == 1)`

```
CASPAL <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Xn|SP>{, #0}]
```

Encoding for the 32-bit CASPL variant Applies when `(sz == 0 && L == 0 && o0 == 1)`

```
CASPL <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit CASP variant Applies when `(sz == 1 && L == 0 && o0 == 0)`

```
CASP <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit CASPA variant Applies when `(sz == 1 && L == 1 && o0 == 0)`

```
CASPA <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit CASPAL variant Applies when `(sz == 1 && L == 1 && o0 == 1)`

```
CASPAL <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit CASPL variant Applies when `(sz == 1 && L == 0 && o0 == 1)`

```
CASPL <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
if Rs<0> == '1' || Rt<0> == '1' then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sz);
constant boolean acquire = L == '1';
constant boolean release = o0 == '1';
```

Assembler Symbols

<Ws> Is the 32-bit name of the first general-purpose register to be compared and loaded, encoded in the "Rs" field. <Ws> must be an even-numbered register.

<W(s+1)> Is the 32-bit name of the second general-purpose register to be compared and loaded.

<Wt> Is the 32-bit name of the first general-purpose register to be conditionally stored, encoded in the "Rt" field. <Wt> must be an even-numbered register.

<W(t+1)> Is the 32-bit name of the second general-purpose register to be conditionally stored.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<**Xs**> Is the 64-bit name of the first general-purpose register to be compared and loaded, encoded in the “Rs” field. <Xs> must be an even-numbered register.

<**X(s+1)**> Is the 64-bit name of the second general-purpose register to be compared and loaded.

<**Xt**> Is the 64-bit name of the first general-purpose register to be conditionally stored, encoded in the “Rt” field. <Xt> must be an even-numbered register.

<**X(t+1)**> Is the 64-bit name of the second general-purpose register to be conditionally stored.

Operation

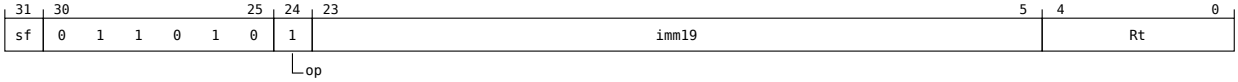
```
bits(64) address;
bits(2*datasize) comparevalue;
bits(2*datasize) newvalue;
bits(2*datasize) data;

constant bits(datasize) s1 = X[s, datasize];
constant bits(datasize) s2 = X[s+1, datasize];
constant bits(datasize) t1 = X[t, datasize];
constant bits(datasize) t2 = X[t+1, datasize];

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_CAS, acquire, release,
                                                          privileged, t, t+1, s, s+1);

comparevalue = if BigEndian(accdesc.acctype) then s1:s2 else s2:s1;
newvalue     = if BigEndian(accdesc.acctype) then t1:t2 else t2:t1;
if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

data = MemAtomic(address, comparevalue, newvalue, accdesc);

if BigEndian(accdesc.acctype) then
    X[s, datasize]   = data<2*datasize-1:datasize>;
    X[s+1, datasize] = data<datasize-1:0>;
else
    X[s, datasize]   = data<datasize-1:0>;
    X[s+1, datasize] = data<2*datasize-1:datasize>;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.41 CBNZ

Compare and branch on nonzero

This instruction compares the value in a register with zero, and conditionally branches to a label at a PC-relative offset if the comparison is not equal. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
CBNZ <Wt>, <label>
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
CBNZ <Xt>, <label>
```

Decode for all variants of this encoding

```
constant integer t = UInt(Rt);
constant integer datasize = 32 << UInt(sf);
constant bits(64) offset = SignExtend(imm19:'00', 64);
```

Assembler Symbols

<**Wt**> Is the 32-bit name of the general-purpose register to be tested, encoded in the "Rt" field.

<**label**> Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

<**Xt**> Is the 64-bit name of the general-purpose register to be tested, encoded in the "Rt" field.

Operation

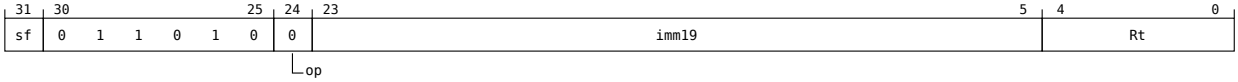
```
constant boolean branch_conditional = TRUE;
constant bits(datasize) operand1 = X[t, datasize];
if !IsZero(operand1) then
    BranchTo(PC64 + offset, BranchType_DIR, branch_conditional);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.42 CBZ

Compare and branch on zero

This instruction compares the value in a register with zero, and conditionally branches to a label at a PC-relative offset if the comparison is equal. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect condition flags.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
CBZ <Wt>, <label>
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
CBZ <Xt>, <label>
```

Decode for all variants of this encoding

```
constant integer t = UInt(Rt);
constant integer datasize = 32 << UInt(sf);
constant bits(64) offset = SignExtend(imm19:'00', 64);
```

Assembler Symbols

<**Wt**> Is the 32-bit name of the general-purpose register to be tested, encoded in the "Rt" field.

<**label**> Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

<**Xt**> Is the 64-bit name of the general-purpose register to be tested, encoded in the "Rt" field.

Operation

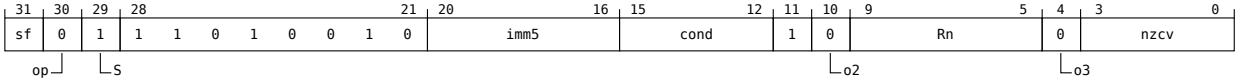
```
constant boolean branch_conditional = TRUE;
constant bits(datasize) operand1 = X[t, datasize];
if IsZero(operand1) then
    BranchTo(PC64 + offset, BranchType_DIR, branch_conditional);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.43 CCMN (immediate)

Conditional compare negative (immediate)

This instruction sets the value of the condition flags to the result of the comparison of a register value and a negated immediate value if the condition is TRUE, and an immediate value otherwise.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
CCMN <Wn>, #<imm>, #<nzcv>, <cond>
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
CCMN <Xn>, #<imm>, #<nzcv>, <cond>
```

Decode for all variants of this encoding

```
constant integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);
constant bits(4) condition = cond;
bits(4) flags = nzcv;
constant bits(datasize) imm = ZeroExtend(imm5, datasize);
```

Assembler Symbols

<**Wn**> Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<**imm**> Is a five bit unsigned (positive) immediate encoded in the “imm5” field.

<**nzcv**> Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the “nzcv” field.

<**cond**> Is one of the standard conditions, encoded in the standard way, and encoded in “cond”:

cond	<cond>
0000	EQ
0001	NE
0010	CS
0011	CC
0100	MI
0101	PL
0110	VS
0111	VC
1000	HI
1001	LS
1010	GE
1011	LT
1100	GT

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

cond	<cond>
1101	LE
1110	AL
1111	NV

<**Xn**> Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

```
if ConditionHolds(condition) then
    constant bits(datasize) operand1 = X[n, datasize];
    constant bits(datasize) operand2 = imm;
    (-, flags) = AddWithCarry(operand1, operand2, '0');
PSTATE.<N,Z,C,V> = flags;
```

Operational information

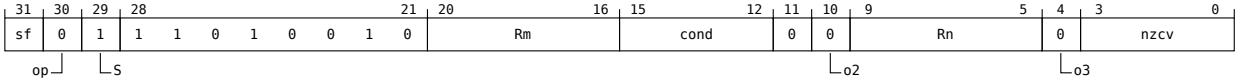
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.44 CCMN (register)

Conditional compare negative (register)

This instruction sets the value of the condition flags to the result of the comparison of a register value and the inverse of another register value if the condition is TRUE, and an immediate value otherwise.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
CCMN <Wn>, <Wm>, #<nzcv>, <cond>
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
CCMN <Xn>, <Xm>, #<nzcv>, <cond>
```

Decode for all variants of this encoding

```
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
constant bits(4) condition = cond;
bits(4) flags = nzcv;
```

Assembler Symbols

<**Wn**> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

<**Wm**> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<**nzcv**> Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field.

<**cond**> Is one of the standard conditions, encoded in the standard way, and encoded in "cond":

cond	<cond>
0000	EQ
0001	NE
0010	CS
0011	CC
0100	MI
0101	PL
0110	VS
0111	VC
1000	HI
1001	LS
1010	GE
1011	LT
1100	GT

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

cond	<cond>
1101	LE
1110	AL
1111	NV

<**Xn**> Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Xm**> Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Operation

```
if ConditionHolds(condition) then
    constant bits(datasize) operand1 = X[n, datasize];
    constant bits(datasize) operand2 = X[m, datasize];
    (-, flags) = AddWithCarry(operand1, operand2, '0');
PSTATE.<N,Z,C,V> = flags;
```

Operational information

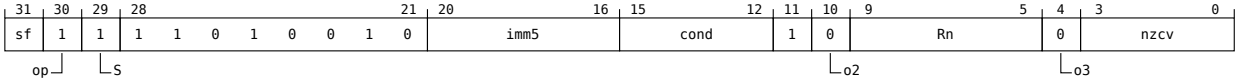
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.45 CCMP (immediate)

Conditional compare (immediate)

This instruction sets the value of the condition flags to the result of the comparison of a register value and an immediate value if the condition is TRUE, and an immediate value otherwise.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
CCMP <Wn>, #<imm>, #<nzcv>, <cond>
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
CCMP <Xn>, #<imm>, #<nzcv>, <cond>
```

Decode for all variants of this encoding

```
constant integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);
constant bits(4) condition = cond;
bits(4) flags = nzcv;
constant bits(datasize) imm = ZeroExtend(imm5, datasize);
```

Assembler Symbols

<**Wn**> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

<**imm**> Is a five bit unsigned (positive) immediate encoded in the "imm5" field.

<**nzcv**> Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field.

<**cond**> Is one of the standard conditions, encoded in the standard way, and encoded in "cond":

cond	<cond>
0000	EQ
0001	NE
0010	CS
0011	CC
0100	MI
0101	PL
0110	VS
0111	VC
1000	HI
1001	LS
1010	GE
1011	LT
1100	GT

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

cond	<cond>
1101	LE
1110	AL
1111	NV

<**Xn**> Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

```
if ConditionHolds(condition) then
    constant bits(datasize) operand1 = X[n, datasize];
    constant bits(datasize) operand2 = imm;
    (-, flags) = AddWithCarry(operand1, NOT(operand2), '1');
PSTATE.<N,Z,C,V> = flags;
```

Operational information

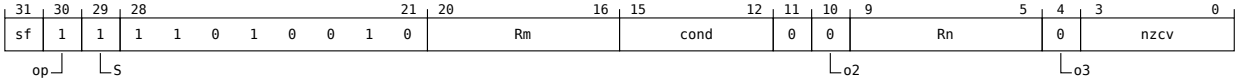
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.46 CCMP (register)

Conditional compare (register)

This instruction sets the value of the condition flags to the result of the comparison of two registers if the condition is TRUE, and an immediate value otherwise.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
CCMP <Wn>, <Wm>, #<nzcv>, <cond>
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
CCMP <Xn>, <Xm>, #<nzcv>, <cond>
```

Decode for all variants of this encoding

```
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
constant bits(4) condition = cond;
bits(4) flags = nzcv;
```

Assembler Symbols

<**Wn**> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

<**Wm**> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<**nzcv**> Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field.

<**cond**> Is one of the standard conditions, encoded in the standard way, and encoded in "cond":

cond	<cond>
0000	EQ
0001	NE
0010	CS
0011	CC
0100	MI
0101	PL
0110	VS
0111	VC
1000	HI
1001	LS
1010	GE
1011	LT
1100	GT

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

cond	<cond>
1101	LE
1110	AL
1111	NV

<**Xn**> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

<**Xm**> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionHolds(condition) then
    constant bits(datasize) operand1 = X[n, datasize];
    constant bits(datasize) operand2 = X[m, datasize];
    (-, flags) = AddWithCarry(operand1, NOT(operand2), '1');
PSTATE.<N,Z,C,V> = flags;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

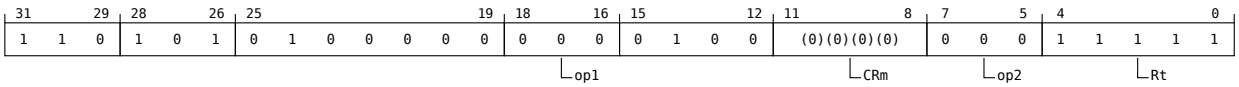
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.47 CFINV

Invert carry flag

This instruction inverts the value of the PSTATE.C flag.

System
(FEAT_FlagM)



Encoding

```
CFINV
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FlagM) then EndOfDecode(Decode_UNDEF);
```

Operation

```
PSTATE.C = NOT(PSTATE.C);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.48 CFP

Control flow prediction restriction by context

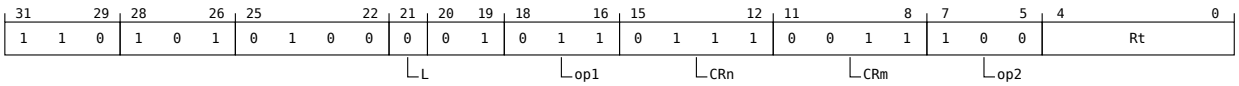
This instruction prevents control flow predictions that predict execution addresses based on information gathered from earlier execution within a particular execution context. Control flow predictions determined by the actions of code in the target execution context or contexts appearing in program order before the instruction cannot be used to exploitatively control speculative execution occurring after the instruction is complete and synchronized.

For more information, see CFP RCTX, Control Flow Prediction Restriction by Context.

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System
(FEAT_SPECRES)



Encoding

```
CFP RCTX, <Xt>
```

is equivalent to

```
SYS #3, C7, C3, #4, <Xt>
```

and is always the preferred disassembly.

Assembler Symbols

<Xt> Is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.

Operation

The description of SYS gives the operational pseudocode for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

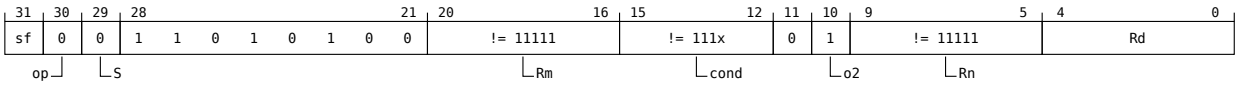
C6.2.49 CINC

Conditional increment

This instruction returns, in the destination register, the value of the source register incremented by 1 if the condition is TRUE, and otherwise returns the value of the source register.

This is an alias of CSINC. This means:

- The encodings in this description are named to match the encodings of CSINC.
- The description of CSINC gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
CINC <Wd>, <Wn>, <invcond>
```

is equivalent to

```
CSINC <Wd>, <Wn>, <Wm>, <cond>
```

and is the preferred disassembly when `Rn == Rm`.

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
CINC <Xd>, <Xn>, <invcond>
```

is equivalent to

```
CSINC <Xd>, <Xn>, <Xm>, <cond>
```

and is the preferred disassembly when `Rn == Rm`.

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Wn**> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.

<**invcond**> Is one of the standard conditions, excluding AL and NV, encoded with its least significant bit inverted, and encoded in "cond":

cond	<invcond>	Description
0000	NE	Maps to <cond> EQ.
0001	EQ	Maps to <cond> NE.
0010	CC	Maps to <cond> CS.
0011	CS	Maps to <cond> CC.
0100	PL	Maps to <cond> MI.
0101	MI	Maps to <cond> PL.
0110	VC	Maps to <cond> VS.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

cond	<invcond>	Description
0111	VS	Maps to <cond> VC.
1000	LS	Maps to <cond> HI.
1001	HI	Maps to <cond> LS.
1010	LT	Maps to <cond> GE.
1011	GE	Maps to <cond> LT.
1100	LE	Maps to <cond> GT.
1101	GT	Maps to <cond> LE.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Xn**> Is the 64-bit name of the general-purpose source register, encoded in the “Rn” and “Rm” fields.

Operation

The description of CSINC gives the operational pseudocode for this instruction.

Operational Information

The description of CSINC gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

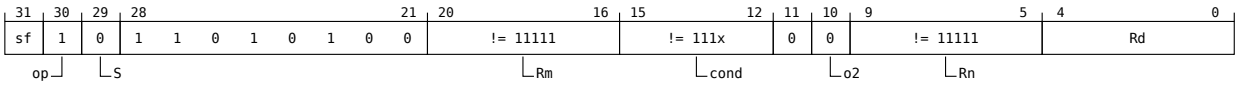
C6.2.50 CINV

Conditional invert

This instruction returns, in the destination register, the bitwise inversion of the value of the source register if the condition is TRUE, and otherwise returns the value of the source register.

This is an alias of CSINV. This means:

- The encodings in this description are named to match the encodings of CSINV.
- The description of CSINV gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
CINV <Wd>, <Wn>, <invcond>
```

is equivalent to

```
CSINV <Wd>, <Wn>, <Wm>, <cond>
```

and is the preferred disassembly when `Rn == Rm`.

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
CINV <Xd>, <Xn>, <invcond>
```

is equivalent to

```
CSINV <Xd>, <Xn>, <Xm>, <cond>
```

and is the preferred disassembly when `Rn == Rm`.

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Wn**> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.

<**invcond**> Is one of the standard conditions, excluding AL and NV, encoded with its least significant bit inverted, and encoded in "cond":

cond	<invcond>	Description
0000	NE	Maps to <cond> EQ.
0001	EQ	Maps to <cond> NE.
0010	CC	Maps to <cond> CS.
0011	CS	Maps to <cond> CC.
0100	PL	Maps to <cond> MI.
0101	MI	Maps to <cond> PL.
0110	VC	Maps to <cond> VS.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

cond	<invcond>	Description
0111	VS	Maps to <cond> VC.
1000	LS	Maps to <cond> HI.
1001	HI	Maps to <cond> LS.
1010	LT	Maps to <cond> GE.
1011	GE	Maps to <cond> LT.
1100	LE	Maps to <cond> GT.
1101	GT	Maps to <cond> LE.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Xn**> Is the 64-bit name of the general-purpose source register, encoded in the “Rn” and “Rm” fields.

Operation

The description of CSINV gives the operational pseudocode for this instruction.

Operational Information

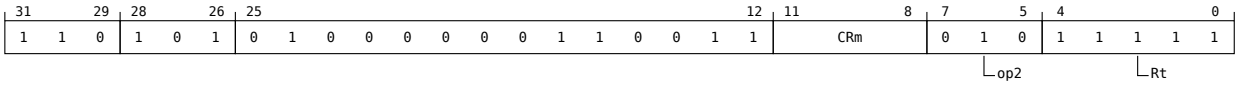
The description of CSINV gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.51 CLREX

Clear exclusive

This instruction clears the local monitor of the executing PE.



Encoding

```
CLREX {#<imm>}
```

Decode for this encoding

```
// CRm field is ignored
```

Assembler Symbols

<**imm**> Is an optional 4-bit unsigned immediate, in the range 0 to 15, defaulting to 15 and encoded in the “CRm” field.

Operation

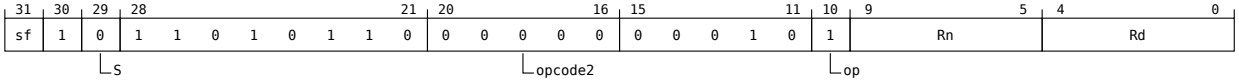
```
ClearExclusiveLocal(ProcessorID());
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.52 CLS

Count leading sign bits

This instruction determines the number of leading bits of the source register that have the same value as the most significant bit of the register, and writes the result to the destination register. The result does not include the most significant bit of the source register.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
CLS <Wd>, <Wn>
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
CLS <Xd>, <Xn>
```

Decode for all variants of this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);
```

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wn**> Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Xn**> Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

```
constant bits(datasize) operand1 = X[n, datasize];
constant integer result = CountLeadingSignBits(operand1);

X[d, datasize] = result<datasize-1:0>;
```

Operational information

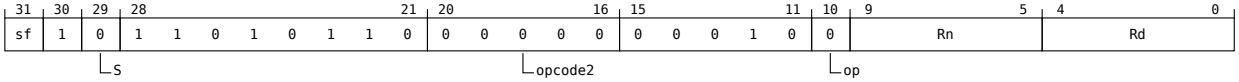
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.53 CLZ

Count leading zeros

This instruction determines the number of consecutive binary zero bits, starting from the most significant bit in the source register, and places the result in the destination register.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
CLZ <Wd>, <Wn>
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
CLZ <Xd>, <Xn>
```

Decode for all variants of this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);
```

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wn**> Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Xn**> Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

```
constant bits(datasize) operand1 = X[n, datasize];
constant integer result = CountLeadingZeroBits(operand1);

X[d, datasize] = result<datasize-1:0>;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

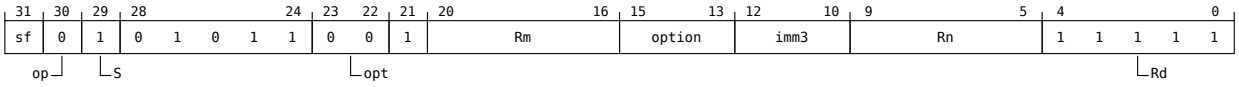
C6.2.54 CMN (extended register)

Compare negative (extended register)

This instruction adds a register value and a sign or zero-extended register value, followed by an optional left shift amount. The argument that is extended from the <Rm> register can be a byte, halfword, word, or doubleword. It updates the condition flags based on the result, and discards the result.

This is an alias of ADDS (extended register). This means:

- The encodings in this description are named to match the encodings of ADDS (extended register).
- The description of ADDS (extended register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
CMN <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

is equivalent to

```
ADDS WZR, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

and is always the preferred disassembly.

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
CMN <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

is equivalent to

```
ADDS XZR, <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

and is always the preferred disassembly.

Assembler Symbols

<Wn|WSP> Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<extend> For the "32-bit" variant: is the extension to be applied to the second source operand, encoded in "option":

option	<extend>
000	UXTB
001	UXTH
010	LSL|UXTW
011	UXTX
100	SXTB
101	SXTH
110	SXTW

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

option	<extend>
111	SXTX

If “Rn” is ’11111’ (WSP) and “option” is ’010’ then LSL is preferred, but may be omitted when “imm3” is ’000’. In all other cases <extend> is required and must be UXTW when “option” is ’010’.

For the “64-bit” variant: is the extension to be applied to the second source operand, encoded in “option”:

option	<extend>
000	UXTB
001	UXTH
010	UXTW
011	LSL|UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

If “Rn” is ’11111’ (SP) and “option” is ’011’ then LSL is preferred, but may be omitted when “imm3” is ’000’. In all other cases <extend> is required and must be UXTX when “option” is ’011’.

<**amount**> Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the “imm3” field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

<**Xn**|**SP**> Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the “Rn” field.

<**R**> Is a width specifier, encoded in “option”:

option	<R>
00x	W
010	W
x11	X
10x	W
110	W

<**m**> Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the “Rm” field.

Operation

The description of ADDS (extended register) gives the operational pseudocode for this instruction.

Operational Information

The description of ADDS (extended register) gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

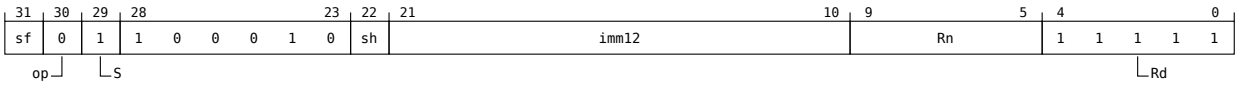
C6.2.55 CMN (immediate)

Compare negative (immediate)

This instruction adds a register value and an optionally-shifted immediate value. It updates the condition flags based on the result, and discards the result.

This is an alias of ADDS (immediate). This means:

- The encodings in this description are named to match the encodings of ADDS (immediate).
- The description of ADDS (immediate) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
CMN <Wn|WSP>, #<imm>{, <shift>}
```

is equivalent to

```
ADDS WZR, <Wn|WSP>, #<imm>{, <shift>}
```

and is always the preferred disassembly.

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
CMN <Xn|SP>, #<imm>{, <shift>}
```

is equivalent to

```
ADDS XZR, <Xn|SP>, #<imm>{, <shift>}
```

and is always the preferred disassembly.

Assembler Symbols

<**Wn|WSP**> Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

<**imm**> Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.

<**shift**> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in "sh":

sh	<shift>
0	LSL #0
1	LSL #12

<**Xn|SP**> Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

Operation

The description of ADDS (immediate) gives the operational pseudocode for this instruction.

Operational Information

The description of ADDS (immediate) gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

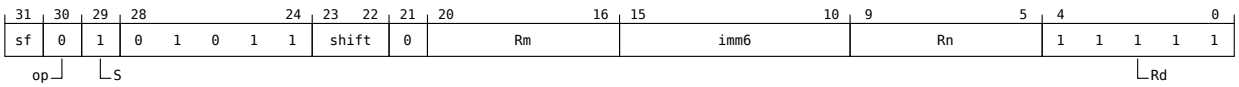
C6.2.56 CMN (shifted register)

Compare negative (shifted register)

This instruction adds a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

This is an alias of ADDS (shifted register). This means:

- The encodings in this description are named to match the encodings of ADDS (shifted register).
- The description of ADDS (shifted register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
CMN <Wn>, <Wm>{, <shift> #<amount>}
```

is equivalent to

```
ADDS WZR, <Wn>, <Wm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
CMN <Xn>, <Xm>{, <shift> #<amount>}
```

is equivalent to

```
ADDS XZR, <Xn>, <Xm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

Assembler Symbols

<**Wn**> Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Wm**> Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<**shift**> Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in “shift”:

shift	<**shift**>
00	LSL
01	LSR
10	ASR
11	RESERVED

<**amount**> For the “32-bit” variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm6” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Operation

The description of ADDS (shifted register) gives the operational pseudocode for this instruction.

Operational Information

The description of ADDS (shifted register) gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

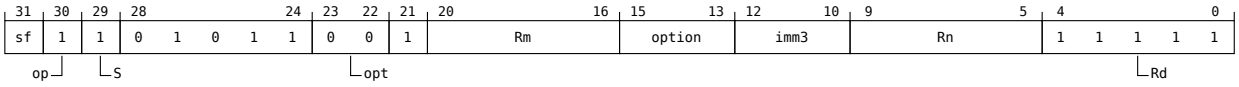
C6.2.57 CMP (extended register)

Compare (extended register)

This instruction subtracts a sign or zero-extended register value, followed by an optional left shift amount, from a register value. The argument that is extended from the <Rm> register can be a byte, halfword, word, or doubleword. It updates the condition flags based on the result, and discards the result.

This is an alias of SUBS (extended register). This means:

- The encodings in this description are named to match the encodings of SUBS (extended register).
- The description of SUBS (extended register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
CMP <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

is equivalent to

```
SUBS WZR, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

and is always the preferred disassembly.

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
CMP <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

is equivalent to

```
SUBS XZR, <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

and is always the preferred disassembly.

Assembler Symbols

<Wn|WSP> Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<extend> For the "32-bit" variant: is the extension to be applied to the second source operand, encoded in "option":

option	<extend>
000	UXTB
001	UXTH
010	LSL|UXTW
011	UXTX
100	SXTB
101	SXTH
110	SXTW

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

option	<extend>
111	SXTX

If “Rn” is ’11111’ (WSP) and “option” is ’010’ then LSL is preferred, but may be omitted when “imm3” is ’000’. In all other cases <extend> is required and must be UXTW when “option” is ’010’.

For the “64-bit” variant: is the extension to be applied to the second source operand, encoded in “option”:

option	<extend>
000	UXTB
001	UXTH
010	UXTW
011	LSL|UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

If “Rn” is ’11111’ (SP) and “option” is ’011’ then LSL is preferred, but may be omitted when “imm3” is ’000’. In all other cases <extend> is required and must be UXTX when “option” is ’011’.

<**amount**> Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the “imm3” field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

<**Xn|SP**> Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the “Rn” field.

<**R**> Is a width specifier, encoded in “option”:

option	<R>
00x	W
010	W
x11	X
10x	W
110	W

<**m**> Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the “Rm” field.

Operation

The description of SUBS (extended register) gives the operational pseudocode for this instruction.

Operational Information

The description of SUBS (extended register) gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

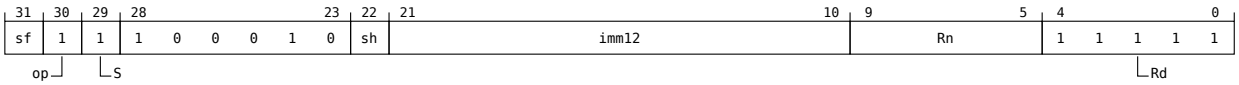
C6.2.58 CMP (immediate)

Compare (immediate)

This instruction subtracts an optionally-shifted immediate value from a register value. It updates the condition flags based on the result, and discards the result.

This is an alias of SUBS (immediate). This means:

- The encodings in this description are named to match the encodings of SUBS (immediate).
- The description of SUBS (immediate) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
CMP <Wn|WSP>, #<imm>{, <shift>}
```

is equivalent to

```
SUBS WZR, <Wn|WSP>, #<imm>{, <shift>}
```

and is always the preferred disassembly.

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
CMP <Xn|SP>, #<imm>{, <shift>}
```

is equivalent to

```
SUBS XZR, <Xn|SP>, #<imm>{, <shift>}
```

and is always the preferred disassembly.

Assembler Symbols

<**Wn**|**WSP**> Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

<**imm**> Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.

<**shift**> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in "sh":

sh	<shift>
0	LSL #0
1	LSL #12

<**Xn**|**SP**> Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

Operation

The description of SUBS (immediate) gives the operational pseudocode for this instruction.

Operational Information

The description of SUBS (immediate) gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

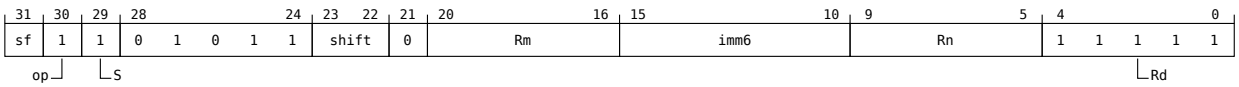
C6.2.59 CMP (shifted register)

Compare (shifted register)

This instruction subtracts an optionally-shifted register value from a register value. It updates the condition flags based on the result, and discards the result.

This is an alias of SUBS (shifted register). This means:

- The encodings in this description are named to match the encodings of SUBS (shifted register).
- The description of SUBS (shifted register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
CMP <Wn>, <Wm>{, <shift> #<amount>}
```

is equivalent to

```
SUBS WZR, <Wn>, <Wm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
CMP <Xn>, <Xm>{, <shift> #<amount>}
```

is equivalent to

```
SUBS XZR, <Xn>, <Xm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

Assembler Symbols

<**Wn**> Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Wm**> Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<**shift**> Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in “shift”:

shift	<**shift**>
00	LSL
01	LSR
10	ASR
11	RESERVED

<**amount**> For the “32-bit” variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm6” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Operation

The description of SUBS (shifted register) gives the operational pseudocode for this instruction.

Operational Information

The description of SUBS (shifted register) gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

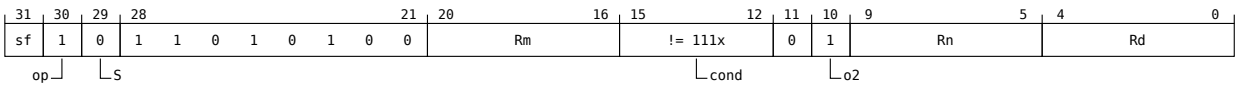
C6.2.60 CNEG

Conditional negate

This instruction returns, in the destination register, the negated value of the source register if the condition is TRUE, and otherwise returns the value of the source register.

This is an alias of CSNEG. This means:

- The encodings in this description are named to match the encodings of CSNEG.
- The description of CSNEG gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
CNEG <Wd>, <Wn>, <invcond>
```

is equivalent to

```
CSNEG <Wd>, <Wn>, <Wm>, <cond>
```

and is the preferred disassembly when `Rn == Rm`.

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
CNEG <Xd>, <Xn>, <invcond>
```

is equivalent to

```
CSNEG <Xd>, <Xn>, <Xm>, <cond>
```

and is the preferred disassembly when `Rn == Rm`.

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Wn**> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.

<**invcond**> Is one of the standard conditions, excluding AL and NV, encoded with its least significant bit inverted, and encoded in "cond":

cond	<invcond>	Description
0000	NE	Maps to <cond> EQ.
0001	EQ	Maps to <cond> NE.
0010	CC	Maps to <cond> CS.
0011	CS	Maps to <cond> CC.
0100	PL	Maps to <cond> MI.
0101	MI	Maps to <cond> PL.
0110	VC	Maps to <cond> VS.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

cond	<invcond>	Description
0111	VS	Maps to <cond> VC.
1000	LS	Maps to <cond> HI.
1001	HI	Maps to <cond> LS.
1010	LT	Maps to <cond> GE.
1011	GE	Maps to <cond> LT.
1100	LE	Maps to <cond> GT.
1101	GT	Maps to <cond> LE.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Xn**> Is the 64-bit name of the general-purpose source register, encoded in the “Rn” and “Rm” fields.

Operation

The description of CSNEG gives the operational pseudocode for this instruction.

Operational Information

The description of CSNEG gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.61 CPP

Cache prefetch prediction restriction by context

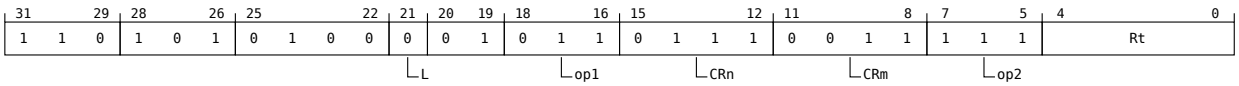
This instruction prevents cache allocation predictions that predict execution addresses based on information gathered from earlier execution within a particular execution context. The actions of code in the target execution context or contexts appearing in program order before the instruction cannot exploitatively control cache prefetch predictions occurring after the instruction is complete and synchronized.

For more information, see CPP RCTX, Cache Prefetch Prediction Restriction by Context.

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System
(FEAT_SPECRES)



Encoding

```
CPP RCTX, <Xt>
```

is equivalent to

```
SYS #3, C7, C3, #7, <Xt>
```

and is always the preferred disassembly.

Assembler Symbols

<Xt> Is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.

Operation

The description of SYS gives the operational pseudocode for this instruction.

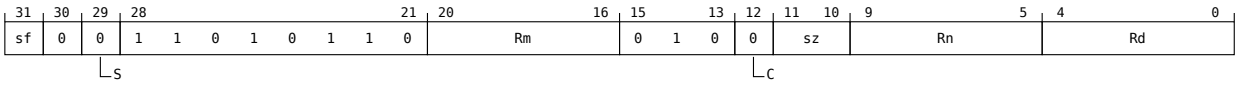
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.62 CRC32B, CRC32H, CRC32W, CRC32X

CRC32 checksum

This instruction performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register. It takes an input CRC value in the first source operand, performs a CRC on the input value in the second source operand, and returns the output CRC value. The second source operand can be 8, 16, 32, or 64 bits. To align with common usage, the bit order of the values is reversed as part of the operation, and the polynomial 0x04C11DB7 is used for the CRC calculation.

CRC
(FEAT_CRC32)



Encoding for the CRC32B variant Applies when `(sf == 0 && sz == 00)`

```
CRC32B <Wd>, <Wn>, <Wm>
```

Encoding for the CRC32H variant Applies when `(sf == 0 && sz == 01)`

```
CRC32H <Wd>, <Wn>, <Wm>
```

Encoding for the CRC32W variant Applies when `(sf == 0 && sz == 10)`

```
CRC32W <Wd>, <Wn>, <Wm>
```

Encoding for the CRC32X variant Applies when `(sf == 1 && sz == 11)`

```
CRC32X <Wd>, <Wn>, <Xm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CRC32) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
if sf == '1' && sz != '11' then EndOfDecode(Decode_UNDEF);
if sf == '0' && sz == '11' then EndOfDecode(Decode_UNDEF);
constant integer size = 8 << UInt(sz);
```

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose accumulator output register, encoded in the “Rd” field.

<Wn> Is the 32-bit name of the general-purpose accumulator input register, encoded in the “Rn” field.

<Wm> Is the 32-bit name of the general-purpose data source register, encoded in the “Rm” field.

<Xm> Is the 64-bit name of the general-purpose data source register, encoded in the “Rm” field.

Operation

```
constant bits(32)      acc     = X[n, 32];     // accumulator
constant bits(size)    val     = X[m, size];   // input value
constant bits(32)      poly    = 0x04C11DB7<31:0>;

constant bits(32+size) tempacc = BitReverse(acc):Zeros(size);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant bits(size+32) tempval = BitReverse(val):Zeros(32);

// Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation
X[d, 32] = BitReverse(Poly32Mod2(tempacc EOR tempval, poly));
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

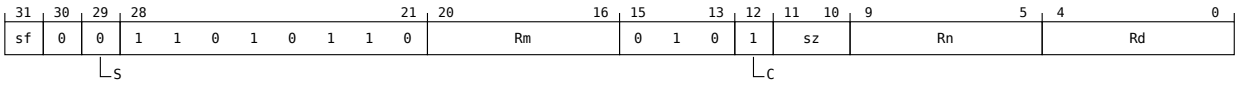
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.63 CRC32CB, CRC32CH, CRC32CW, CRC32CX

CRC32C checksum

This instruction performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register. It takes an input CRC value in the first source operand, performs a CRC on the input value in the second source operand, and returns the output CRC value. The second source operand can be 8, 16, 32, or 64 bits. To align with common usage, the bit order of the values is reversed as part of the operation, and the polynomial 0x1EDC6F41 is used for the CRC calculation.

CRC
(FEAT_CRC32)



Encoding for the CRC32CB variant Applies when `(sf == 0 && sz == 00)`

```
CRC32CB <Wd>, <Wn>, <Wm>
```

Encoding for the CRC32CH variant Applies when `(sf == 0 && sz == 01)`

```
CRC32CH <Wd>, <Wn>, <Wm>
```

Encoding for the CRC32CW variant Applies when `(sf == 0 && sz == 10)`

```
CRC32CW <Wd>, <Wn>, <Wm>
```

Encoding for the CRC32CX variant Applies when `(sf == 1 && sz == 11)`

```
CRC32CX <Wd>, <Wn>, <Xm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CRC32) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
if sf == '1' && sz != '11' then EndOfDecode(Decode_UNDEF);
if sf == '0' && sz == '11' then EndOfDecode(Decode_UNDEF);
constant integer size = 8 << UInt(sz);
```

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose accumulator output register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the general-purpose accumulator input register, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the general-purpose data source register, encoded in the "Rm" field.

<Xm> Is the 64-bit name of the general-purpose data source register, encoded in the "Rm" field.

Operation

```
constant bits(32)      acc     = X[n, 32];     // accumulator
constant bits(size)    val     = X[m, size];   // input value
constant bits(32)      poly    = 0x1EDC6F41<31:0>;

constant bits(32+size) tempacc = BitReverse(acc):Zeros(size);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant bits(size+32) tempval = BitReverse(val):Zeros(32);

// Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation
X[d, 32] = BitReverse(Poly32Mod2(tempacc EOR tempval, poly));
```

Operational information

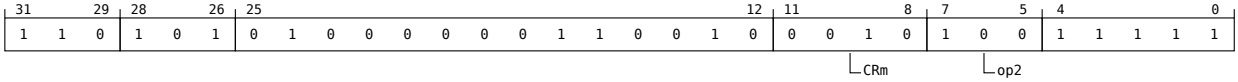
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.64 CSDB

Consumption of speculative data barrier

This instruction is a memory barrier that controls speculative execution arising from data value prediction. For more information and details of the semantics, see Consumption of Speculative Data Barrier (CSDB).



Encoding

CSDB

Decode for this encoding

```
// Empty.
```

Operation

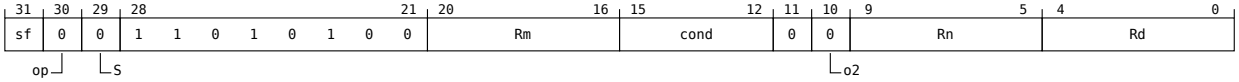
```
ConsumptionOfSpeculativeDataBarrier();
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.65 CSEL

Conditional select

This instruction writes the value of the first source register to the destination register if the condition is TRUE. If the condition is FALSE, it writes the value of the second source register to the destination register.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
CSEL <Wd>, <Wn>, <Wm>, <cond>
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
CSEL <Xd>, <Xn>, <Xm>, <cond>
```

Decode for all variants of this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
constant bits(4) condition = cond;
```

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wn**> Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Wm**> Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<**cond**> Is one of the standard conditions, encoded in the standard way, and encoded in “cond”:

cond	<cond>
0000	EQ
0001	NE
0010	CS
0011	CC
0100	MI
0101	PL
0110	VS
0111	VC
1000	HI
1001	LS
1010	GE
1011	LT
1100	GT

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

cond	<cond>
1101	LE
1110	AL
1111	NV

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Xn**> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

<**Xm**> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Operation

```
bits(datasize) result;
if ConditionHolds(condition) then
    result = X[n, datasize];
else
    result = X[m, datasize];

X[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

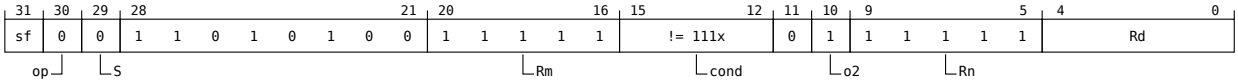
C6.2.66 CSET

Conditional set

This instruction sets the destination register to 1 if the condition is TRUE, and otherwise sets it to 0.

This is an alias of CSINC. This means:

- The encodings in this description are named to match the encodings of CSINC.
- The description of CSINC gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
CSET <Wd>, <invcond>
```

is equivalent to

```
CSINC <Wd>, WZR, WZR, <cond>
```

and is always the preferred disassembly.

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
CSET <Xd>, <invcond>
```

is equivalent to

```
CSINC <Xd>, XZR, XZR, <cond>
```

and is always the preferred disassembly.

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**invcond**> Is one of the standard conditions, excluding AL and NV, encoded with its least significant bit inverted, and encoded in "cond":

cond	<invcond>	Description
0000	NE	Maps to <cond> EQ.
0001	EQ	Maps to <cond> NE.
0010	CC	Maps to <cond> CS.
0011	CS	Maps to <cond> CC.
0100	PL	Maps to <cond> MI.
0101	MI	Maps to <cond> PL.
0110	VC	Maps to <cond> VS.
0111	VS	Maps to <cond> VC.
1000	LS	Maps to <cond> HI.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

cond	<invcond>	Description
1001	HI	Maps to <cond> LS.
1010	LT	Maps to <cond> GE.
1011	GE	Maps to <cond> LT.
1100	LE	Maps to <cond> GT.
1101	GT	Maps to <cond> LE.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

Operation

The description of CSINC gives the operational pseudocode for this instruction.

Operational Information

The description of CSINC gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

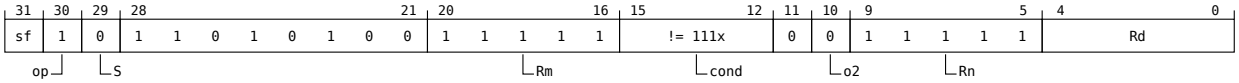
C6.2.67 CSETM

Conditional set mask

This instruction sets all bits of the destination register to 1 if the condition is TRUE, and otherwise sets all bits to 0.

This is an alias of CSINV. This means:

- The encodings in this description are named to match the encodings of CSINV.
- The description of CSINV gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
CSETM <Wd>, <invcond>
```

is equivalent to

```
CSINV <Wd>, WZR, WZR, <cond>
```

and is always the preferred disassembly.

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
CSETM <Xd>, <invcond>
```

is equivalent to

```
CSINV <Xd>, XZR, XZR, <cond>
```

and is always the preferred disassembly.

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**invcond**> Is one of the standard conditions, excluding AL and NV, encoded with its least significant bit inverted, and encoded in "cond":

cond	<invcond>	Description
0000	NE	Maps to <cond> EQ.
0001	EQ	Maps to <cond> NE.
0010	CC	Maps to <cond> CS.
0011	CS	Maps to <cond> CC.
0100	PL	Maps to <cond> MI.
0101	MI	Maps to <cond> PL.
0110	VC	Maps to <cond> VS.
0111	VS	Maps to <cond> VC.
1000	LS	Maps to <cond> HI.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

cond	<invcond>	Description
1001	HI	Maps to <cond> LS.
1010	LT	Maps to <cond> GE.
1011	GE	Maps to <cond> LT.
1100	LE	Maps to <cond> GT.
1101	GT	Maps to <cond> LE.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

Operation

The description of CSINV gives the operational pseudocode for this instruction.

Operational Information

The description of CSINV gives the operational information for this instruction.

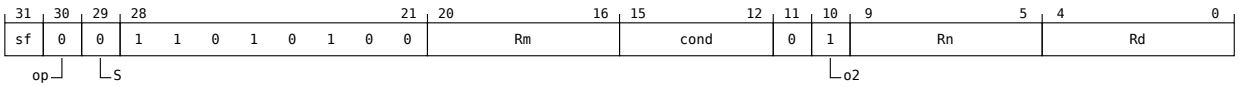
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.68 CSINC

Conditional select increment

This instruction returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the value of the second source register incremented by 1.

This instruction is used by the aliases CINC, and CSET.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
CSINC <Wd>, <Wn>, <Wm>, <cond>
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
CSINC <Xd>, <Xn>, <Xm>, <cond>
```

Decode for all variants of this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
constant bits(4) condition = cond;
```

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wn**> Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Wm**> Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<**cond**> Is one of the standard conditions, encoded in the standard way, and encoded in “cond”:

cond	<cond>
0000	EQ
0001	NE
0010	CS
0011	CC
0100	MI
0101	PL
0110	VS
0111	VC
1000	HI
1001	LS
1010	GE
1011	LT

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

cond	<cond>
1100	GT
1101	LE
1110	AL
1111	NV

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Xn**> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

<**Xm**> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Alias Conditions

Alias	Is preferred when
CINC	Rm != '11111' && Rn != '11111' && !(cond IN {'111x'}) && Rn == Rm
CSET	Rm == '11111' && Rn == '11111' && !(cond IN {'111x'})

Operation

```
bits(datasize) result;
if ConditionHolds(condition) then
    result = X[n, datasize];
else
    result = X[m, datasize] + 1;

X[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

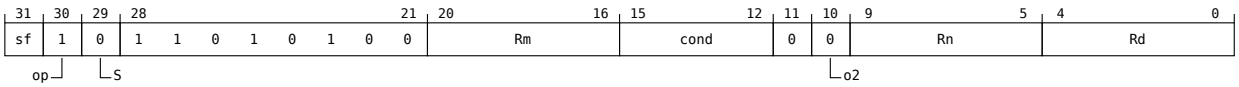
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.69 CSINV

Conditional select invert

This instruction returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the bitwise inversion value of the second source register.

This instruction is used by the aliases CINV, and CSETM.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
CSINV <Wd>, <Wn>, <Wm>, <cond>
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
CSINV <Xd>, <Xn>, <Xm>, <cond>
```

Decode for all variants of this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
constant bits(4) condition = cond;
```

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wn**> Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Wm**> Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<**cond**> Is one of the standard conditions, encoded in the standard way, and encoded in “cond”:

cond	<cond>
0000	EQ
0001	NE
0010	CS
0011	CC
0100	MI
0101	PL
0110	VS
0111	VC
1000	HI
1001	LS
1010	GE
1011	LT

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

cond	<cond>
1100	GT
1101	LE
1110	AL
1111	NV

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Xn**> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

<**Xm**> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Alias Conditions

Alias	Is preferred when
CINV	Rm != '11111' && Rn != '11111' && !(cond IN {'111x'}) && Rn == Rm
CSETM	Rm == '11111' && Rn == '11111' && !(cond IN {'111x'})

Operation

```
bits(datasize) result;
if ConditionHolds(condition) then
    result = X[n, datasize];
else
    result = NOT(X[m, datasize]);

X[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

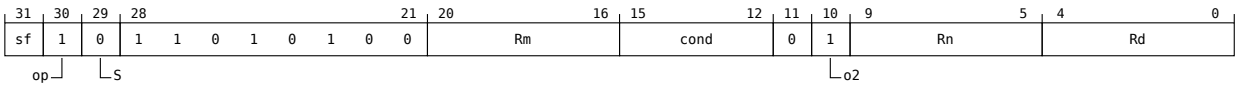
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.70 CSNEG

Conditional select negation

This instruction returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the negated value of the second source register.

This instruction is used by the alias CNEG.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
CSNEG <Wd>, <Wn>, <Wm>, <cond>
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
CSNEG <Xd>, <Xn>, <Xm>, <cond>
```

Decode for all variants of this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
constant bits(4) condition = cond;
```

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Wn**> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

<**Wm**> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<**cond**> Is one of the standard conditions, encoded in the standard way, and encoded in "cond":

cond	<cond>
0000	EQ
0001	NE
0010	CS
0011	CC
0100	MI
0101	PL
0110	VS
0111	VC
1000	HI
1001	LS
1010	GE
1011	LT

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

cond	<cond>
1100	GT
1101	LE
1110	AL
1111	NV

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Xn**> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

<**Xm**> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Alias Conditions

Alias	Is preferred when
CNEG	!(cond IN {'111x'}) && Rn == Rm

Operation

```
bits(datasize) result;
if ConditionHolds(condition) then
    result = X[n, datasize];
else
    result = NOT(X[m, datasize]) + 1;

X[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

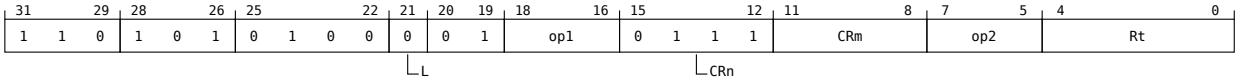
C6.2.71 DC

Data cache operation

For more information, see op0==0b01, cache maintenance, TLB maintenance, and address translation instructions.

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding

```
DC <dc_op>, <Xt>
```

is equivalent to

```
SYS #<op1>, C7, <Cm>, #<op2>, <Xt>
```

and is the preferred disassembly when `SysOp(op1, '0111', CRm, op2) == Sys_DC`.

Assembler Symbols

<**dc_op**> Is a DC operation name, as listed for the DC system instruction group, encoded in "op1:CRm:op2":

op1	CRm	op2	<dc_op>	Architectural Feature
000	0110	001	IVAC	-
000	0110	010	ISW	-
000	1010	010	CSW	-
000	1110	010	CISW	-
011	0100	001	ZVA	-
011	1010	001	CVAC	-
011	1011	001	CVAU	-
011	1100	001	CVAP	FEAT_DPB
011	1101	001	CVADP	FEAT_DPB2
011	1110	001	CIVAC	-

<**Xt**> Is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.

Operation

The description of SYS gives the operational pseudocode for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.72 DCPS1

Debug change PE state to EL1

This instruction, when executed in Debug state:

- If executed at EL0, changes the current Exception level and SP to EL1 using SP_EL1.
- Otherwise, if executed at ELx, selects SP_ELx.

The target Exception level of a DCPS1 instruction is:

- EL1 if the instruction is executed at EL0.
- Otherwise, the Exception level at which the instruction is executed.

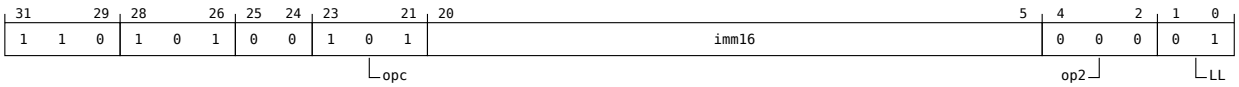
When the target Exception level of a DCPS1 instruction is ELx, on executing this instruction:

- ELR_ELx becomes UNKNOWN.
- SPSR_ELx becomes UNKNOWN.
- ESR_ELx becomes UNKNOWN.
- DLR_EL0 and DSPSR_EL0 become UNKNOWN.
- The endianness is set according to SCTLR_ELx.EE.

This instruction is always UNDEFINED in Non-debug state.

This instruction is UNDEFINED at EL0 if HCR_EL2.TGE == 1.

For more information on the operation of the DCPS<n> instructions, see DCPS.



Encoding

```
DCPS1 {#<imm>}
```

Decode for this encoding

```
// Empty.
```

Assembler Symbols

<imm> Is an optional 16-bit unsigned immediate, in the range 0 to 65535, defaulting to 0 and encoded in the "imm16" field.

Operation

```
if !Halted() then UNDEFINED;
DCPSInstruction(EL1);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.73 DCPS2

Debug change PE state to EL2

This instruction, when executed in Debug state:

- If executed at EL0 or EL1, changes the current Exception level and SP to EL2 using SP_EL2.
- Otherwise, if executed at ELx, selects SP_ELx.

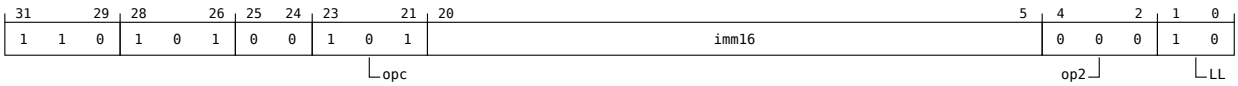
The target Exception level of a DCPS2 instruction is EL2.

When the target Exception level of a DCPS2 instruction is ELx, on executing this instruction:

- ELR_ELx becomes UNKNOWN.
- SPSR_ELx becomes UNKNOWN.
- ESR_ELx becomes UNKNOWN.
- DLR_EL0 and DSPSR_EL0 become UNKNOWN.
- The endianness is set according to SCTLR_ELx.EE.

This instruction is always UNDEFINED in Non-debug state.

For more information on the operation of the DCPS<n> instructions, see DCPS.



Encoding

```
DCPS2 {#<imm>}
```

Decode for this encoding

```
// Empty.
```

Assembler Symbols

<imm> Is an optional 16-bit unsigned immediate, in the range 0 to 65535, defaulting to 0 and encoded in the "imm16" field.

Operation

```
if !Halted() then UNDEFINED;
DCPSInstruction(EL2);
```

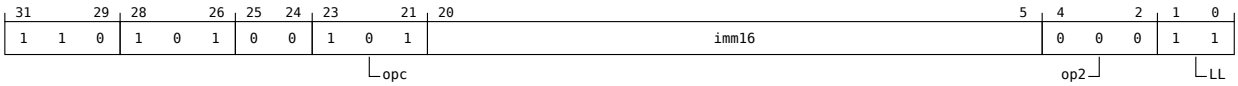

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.74 DCPS3

Debug change PE state to EL3

This instruction is UNDEFINED.

For more information on the operation of the DCPS<n> instructions, see DCPS.



Encoding

```
DCPS3 {#<imm>}
```

Decode for this encoding

```
// Empty.
```

Assembler Symbols

<imm> Is an optional 16-bit unsigned immediate, in the range 0 to 65535, defaulting to 0 and encoded in the "imm16" field.

Operation

```
if !Halted() then UNDEFINED;
DCPSInstruction(EL3);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

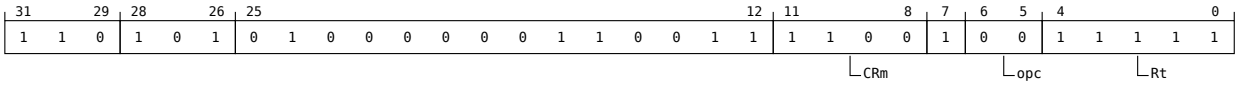
C6.2.75 DFB

Data full barrier

This instruction is a memory barrier that ensures the completion of memory accesses. If executed at EL2, this instruction orders memory accesses irrespective of their Exception level or associated VMID. If executed at EL1 or EL0, this instruction behaves as DSB SY.

This is an alias of DSB. This means:

- The encodings in this description are named to match the encodings of DSB.
- The description of DSB gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding

```
DFB
```

is equivalent to

```
DSB #12
```

and is always the preferred disassembly.

Operation

The description of DSB gives the operational pseudocode for this instruction.

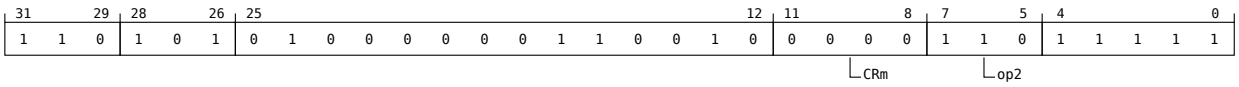
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.76 DGH

Data gathering hint

This instruction is a hint instruction that indicates that it is not expected to be performance optimal to merge memory accesses with Normal Non-cacheable or Device-GRE attributes appearing in program order before the hint instruction with any memory accesses appearing after the hint instruction into a single memory transaction on an interconnect.

System
(FEAT_DGH)



Encoding

DGH

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_DGH) then EndOfDecode(Decode_NOP);
```

Operation

```
Hint_DGH();
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

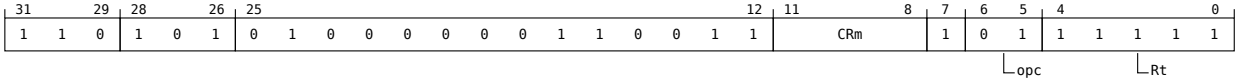
C6.2.77 DMB

Data memory barrier

This instruction is a memory barrier that ensures the ordering of observations of memory accesses, see Data Memory Barrier.

The ordering requirements of Data Memory Barrier instruction is as follows:

- EL1 and EL0 memory accesses are ordered only with respect to memory accesses using the same VMID.
- EL2 memory accesses are ordered only with respect to other EL2 memory accesses.



Encoding

```
DMB (<option>|#<imm>)
```

Decode for this encoding

```
MBReqDomain domain;
MBReqTypes types;
case CRm<3:2> of
    when '00' domain = MBReqDomain_OuterShareable;
    when '01' domain = MBReqDomain_Nonshareable;
    when '10' domain = MBReqDomain_InnerShareable;
    when '11' domain = MBReqDomain_FullSystem;
case CRm<1:0> of
    when '00' types = MBReqTypes_All; domain = MBReqDomain_FullSystem;
    when '01' types = MBReqTypes_Reads;
    when '10' types = MBReqTypes_Writes;
    when '11' types = MBReqTypes_All;

boolean vmid_sensitive = CRm<1:0> != '00';
```

Assembler Symbols

<**option**> Specifies the limitation on the barrier operation. Values are:

SY Full system is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. This option is referred to as the full system barrier. Encoded as CRm = 0b1111.

ST Full system is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b1110.

LD Full system is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b1101.

ISH Inner Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = 0b1011.

ISHST Inner Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b1010.

ISHLD Inner Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b1001.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NSH Non-shareable is the required shareability domain, reads and writes are the required access, both before and after the barrier instruction. Encoded as CRm = 0b0111.

NSHST Non-shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b0110.

NSHLD Non-shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b0101.

OSH Outer Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = 0b0011.

OSHST Outer Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b0010.

OSHLD Outer Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b0001.

All other encodings of "CRm" that are not listed are reserved and can be encoded using the #<imm> syntax. All unsupported and reserved options must execute as a full system barrier operation, but software must not rely on this behavior. For more information on whether an access is before or after a barrier instruction, see Data Memory Barrier (DMB) or see Data Synchronization Barrier (DSB).

CRm	<option>
xx00	RESERVED
0001	OSHLD
0010	OSHST
0011	OSH
0101	NSHLD
0110	NSHST
0111	NSH
1001	ISHLD
1010	ISHST
1011	ISH
1101	LD
1110	ST
1111	SY

<imm> Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "CRm" field.

Operation

```
vmid_sensitive = vmid_sensitive && PSTATE.EL != EL2;
DataMemoryBarrier(domain, types, vmid_sensitive);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.78 DRPS

Debug restore PE state

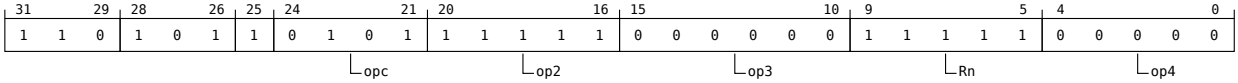
This instruction restores PSTATE from the SPSR.

The SPSR is checked for the current Exception level for an illegal return event. See Illegal exception returns from AArch64 state.

This instruction is UNDEFINED in Non-debug state.

This instruction is UNDEFINED at EL0.

For more information on the operation of DRPS, see DRPS.



Encoding

DRPS

Decode for this encoding

```
// Empty.
```

Operation

```
if !Halted() || PSTATE.EL == EL0 then UNDEFINED;
DRPSInstruction();
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.79 DSB

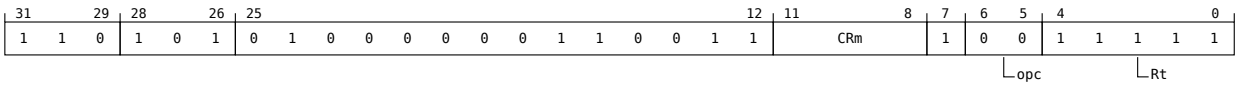
Data synchronization barrier

This instruction is a memory barrier that ensures the completion of memory accesses, see Data Synchronization Barrier.

The ordering requirements of Data Synchronization Barrier instruction is as follows:

- EL1 and EL0 memory accesses are ordered only with respect to memory accesses using the same VMID.
- EL2 memory accesses are ordered only with respect to other EL2 memory accesses.

This instruction is used by the aliases DFB, PSSBB, and SSBB.



Encoding

```
DSB (<option>|#<imm>)
```

Decode for this encoding

```
constant boolean nXS = FALSE;

MBReqDomain domain;
case CRm<3:2> of
    when '00' domain = MBReqDomain_OuterShareable;
    when '01' domain = MBReqDomain_Nonshareable;
    when '10' domain = MBReqDomain_InnerShareable;
    when '11' domain = MBReqDomain_FullSystem;

MBReqTypes types;
case CRm<1:0> of
    when '00' types = MBReqTypes_All; domain = MBReqDomain_FullSystem;
    when '01' types = MBReqTypes_Reads;
    when '10' types = MBReqTypes_Writes;
    when '11' types = MBReqTypes_All;
```

Assembler Symbols

<**option**> Specifies the limitation on the barrier operation. Values are:

SY Full system is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. This option is referred to as the full system barrier. Encoded as CRm = 0b1111.

ST Full system is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b1110.

LD Full system is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b1101.

ISH Inner Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = 0b1011.

ISHST Inner Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b1010.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ISHLD Inner Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b1001.

NSH Non-shareable is the required shareability domain, reads and writes are the required access, both before and after the barrier instruction. Encoded as CRm = 0b0111.

NSHST Non-shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b0110.

NSHLD Non-shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b0101.

OSH Outer Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = 0b0011.

OSHST Outer Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b0010.

OSHLD Outer Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b0001.

All other encodings of "CRm" that are not listed, other than the values 0b0000, 0b0100, and 0b1100, are reserved and can be encoded using the #<imm> syntax. All unsupported and reserved options must execute as a full system barrier operation, but software must not rely on this behavior. For more information on whether an access is before or after a barrier instruction, see Data Memory Barrier (DMB) or see Data Synchronization Barrier (DSB).

Note

The value 0b0000 is used to encode SSBB, the value 0b0100 is used to encode PSSBB, and the value 0b1100 is used to encode DFB.

CRm	<option>
0001	OSHLD
0010	OSHST
0011	OSH
0101	NSHLD
0110	NSHST
0111	NSH
1000	RESERVED
1001	ISHLD
1010	ISHST
1011	ISH
1101	LD
1110	ST
1111	SY

<**imm**> Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "CRm" field.

Alias Conditions

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias	Is preferred when
DFB	CRm == '1100'
PSSBB	CRm == '0100'
SSBB	CRm == '0000'

Operation

```
constant boolean vmid_sensitive = (PSTATE.EL != EL2) && (CRm<1:0> != '00');
DataSynchronizationBarrier(domain, types, vmid_sensitive);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.80 DVP

Data value prediction restriction by context

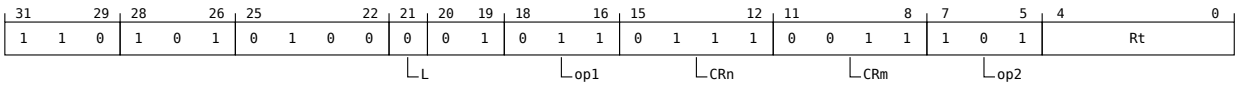
This instruction prevents data value predictions that predict execution addresses based on information gathered from earlier execution within a particular execution context. Data value predictions determined by the actions of code in the target execution context or contexts appearing in program order before the instruction cannot be used to exploitatively control speculative execution occurring after the instruction is complete and synchronized.

For more information, see DVP RCTX, Data Value Prediction Restriction by Context.

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System
(FEAT_SPECRES)



Encoding

```
DVP RCTX, <Xt>
```

is equivalent to

```
SYS #3, C7, C3, #5, <Xt>
```

and is always the preferred disassembly.

Assembler Symbols

<Xt> Is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.

Operation

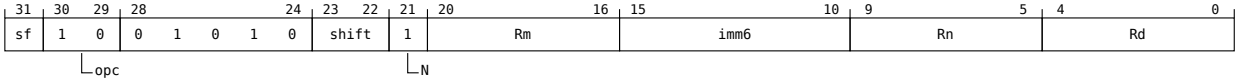
The description of SYS gives the operational pseudocode for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.81 EON (shifted register)

Bitwise exclusive-OR NOT (shifted register)

This instruction performs a bitwise exclusive-OR NOT of a register value and an optionally-shifted register value, and writes the result to the destination register.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
EON <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
EON <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

Decode for all variants of this encoding

```
if sf == '0' && imm6<5> == '1' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
constant ShiftType shift_type = DecodeShift(shift);
constant integer shift_amount = UInt(imm6);
```

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wn**> Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Wm**> Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<**shift**> Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:

shift	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<**amount**> For the “32-bit” variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm6” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Xn**> Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Xm**> Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
constant bits(datasize) operand1 = X[n, datasize];
constant bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount, datasize);

X[d, datasize] = operand1 EOR NOT(operand2);
```

Operational information

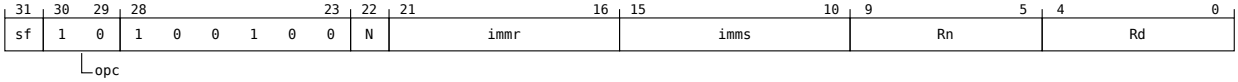
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.82 EOR (immediate)

Bitwise exclusive-OR (immediate)

This instruction performs a bitwise exclusive-OR of a register value and an immediate value, and writes the result to the destination register.



Encoding for the 32-bit variant Applies when `(sf == 0 && N == 0)`

```
EOR <Wd|WSP>, <Wn>, #<imm>
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
EOR <Xd|SP>, <Xn>, #<imm>
```

Decode for all variants of this encoding

```
if sf == '0' && N != '0' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);
bits(datasize) imm;
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE, datasize);
```

Assembler Symbols

<**Wd|WSP**> Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

<**Wn**> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

<**imm**> For the "32-bit" variant: is the bitmask immediate, encoded in "imms:immr".

For the "64-bit" variant: is the bitmask immediate, encoded in "N:imms:immr".

<**Xd|SP**> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

<**Xn**> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

```
constant bits(datasize) operand1 = X[n, datasize];
constant bits(datasize) operand2 = imm;

constant bits(datasize) result = operand1 EOR operand2;

if d == 31 then
    SP[64] = ZeroExtend(result, 64);
else
    X[d, datasize] = result;
```

Operational information

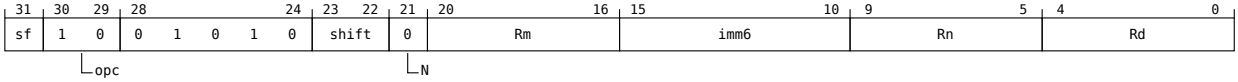
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.83 EOR (shifted register)

Bitwise exclusive-OR (shifted register)

This instruction performs a bitwise exclusive-OR of a register value and an optionally-shifted register value, and writes the result to the destination register.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
EOR <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
EOR <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

Decode for all variants of this encoding

```
if sf == '0' && imm6<5> == '1' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
constant ShiftType shift_type = DecodeShift(shift);
constant integer shift_amount = UInt(imm6);
```

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wn**> Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Wm**> Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<**shift**> Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:

shift	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<**amount**> For the “32-bit” variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm6” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Xn**> Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Xm**> Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
constant bits(datasize) operand1 = X[n, datasize];
constant bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount, datasize);

X[d, datasize] = operand1 EOR operand2;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

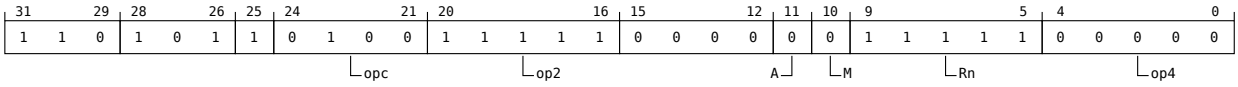
C6.2.84 ERET

Exception return

This instruction restores PSTATE from the SPSR, and branches to the address held in the ELR.

The SPSR is checked for the current Exception level for an illegal return event. See Illegal exception returns from AArch64 state.

ERET is UNDEFINED at EL0.



Encoding

ERET

Decode for this encoding

```
constant boolean pac = FALSE;
constant boolean use_key_a = TRUE;
```

Operation

```
if PSTATE.EL == EL0 then UNDEFINED;
constant bits(64) target = ELR_ELx[];

AArch64.ExceptionReturn(target, SPSR_ELx[]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.85 ERETAA, ERETAB

Exception return, with pointer authentication

This instruction authenticates the address in ELR, using SP as the modifier and the specified key, restores PSTATE from the SPSR for the current Exception level, and branches to the authenticated address.

Key A is used for `ERETAA`. Key B is used for `ERETAB`.

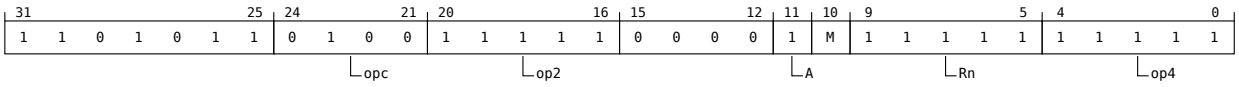
If the authentication passes, the PE continues execution at the target of the branch. For information on behavior if the authentication fails, see Faulting on pointer authentication.

The authenticated address is not written back to ELR.

The SPSR is checked for the current Exception level for an illegal return event. See Illegal exception returns from AArch64 state.

`ERETAA` and `ERETAB` are UNDEFINED at EL0.

Integer
(FEAT_PAuth)



Encoding for the ERETAA variant Applies when `(M == 0)`

`ERETAA`

Encoding for the ERETAB variant Applies when `(M == 1)`

`ERETAB`

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_UNDEF);

constant boolean pac = TRUE;
constant boolean use_key_a = (M == '0');
constant boolean auth_then_branch = TRUE;
```

Operation

```
if PSTATE.EL == EL0 then UNDEFINED;
bits(64) target = ELR_ELx[];
constant bits(64) modifier = SP[64];

if use_key_a then
    target = AuthIA(target, modifier, auth_then_branch);
else
    target = AuthIB(target, modifier, auth_then_branch);

AArch64.ExceptionReturn(target, SPSR_ELx[]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.86 ESB

Error synchronization barrier

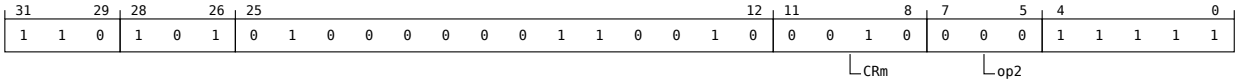
This instruction is an error synchronization event that might also update DISR_EL1 and VDISR_EL2.

This instruction can be used at all Exception levels and in Debug state.

In Debug state, this instruction behaves as if SError interrupts are masked at all Exception levels. For more information, see RAS PE architecture and Arm® Reliability, Availability, and Serviceability (RAS) System Architecture, for A-profile architecture (ARM IHI 0100).

If FEAT_RAS is not implemented, this instruction executes as a `NOP`.

System
(FEAT_RAS)



Encoding

```
ESB
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_RAS) then EndOfDecode(Decode_NOP);
```

Operation

```
SynchronizeErrors();
AArch64.ESBOperation();
if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
    AArch64.vESBOperation();
TakeUnmaskedSErrorInterrupts();
```

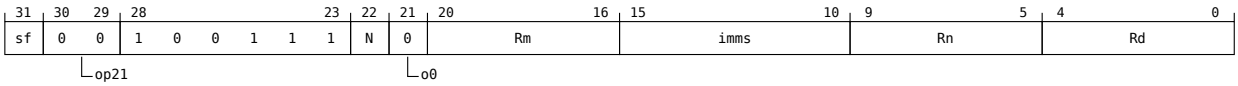

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.87 EXTR

Extract register

This instruction extracts a register from a pair of registers.

This instruction is used by the alias ROR (immediate).



Encoding for the 32-bit variant Applies when `(sf == 0 && N == 0 && imms == 0xxxxx)`

```
EXTR <Wd>, <Wn>, <Wm>, #<lsb>
```

Encoding for the 64-bit variant Applies when `(sf == 1 && N == 1)`

```
EXTR <Xd>, <Xn>, <Xm>, #<lsb>
```

Decode for all variants of this encoding

```
if N != sf then EndOfDecode(Decode_UNDEF);
if sf == '0' && imms<5> == '1' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
constant integer lsb = UInt(imms);
```

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wn**> Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Wm**> Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<**lsb**> For the “32-bit” variant: is the least significant bit position from which to extract, in the range 0 to 31, encoded in the “imms” field.

For the “64-bit” variant: is the least significant bit position from which to extract, in the range 0 to 63, encoded in the “imms” field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Xn**> Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Xm**> Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Alias Conditions

Alias	Is preferred when
ROR (immediate)	Rn == Rm

Operation

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
bits(datasize) result;
constant bits(datasize) operand1 = X[n, datasize];
constant bits(datasize) operand2 = X[m, datasize];
constant bits(2*datasize) concat = operand1:operand2;

result = concat<(lsb+datasize)-1:lsb>;

X[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

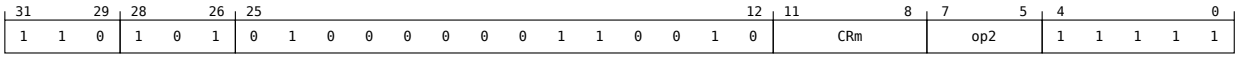
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.88 HINT

Hint instruction

This instruction is for the instruction set space that is reserved for architectural hint instructions.

Some encodings described here are not allocated in this revision of the architecture, and behave as NOPs. These encodings might be allocated to other hint functionality in future revisions of the architecture and therefore must not be used by software.



Encoding

```
HINT #<imm>
```

Decode for this encoding

```
SystemHintOp op;

case CRm:op2 of
    when '0000 000' op = SystemHintOp_NOP;
    when '0000 001' op = SystemHintOp_YIELD;
    when '0000 010' op = SystemHintOp_WFE;
    when '0000 011' op = SystemHintOp_WFI;
    when '0000 100' op = SystemHintOp_SEV;
    when '0000 101' op = SystemHintOp_SEVL;
    when '0000 110'
        if !IsFeatureImplemented(FEAT_DGH) then EndOfDecode(Decode_NOP);
        op = SystemHintOp_DGH;
    when '0000 111' SEE "XPACLRI";
    when '0001 xxx'
        case op2 of
            when '000' SEE "PACIA1716";
            when '010' SEE "PACIB1716";
            when '100' SEE "AUTIA1716";
            when '110' SEE "AUTIB1716";
            otherwise EndOfDecode(Decode_NOP);
    when '0010 000'
        if !IsFeatureImplemented(FEAT_RAS) then EndOfDecode(Decode_NOP);
        op = SystemHintOp_ESB;
    when '0010 010'
        if !IsFeatureImplemented(FEAT_TRF) then EndOfDecode(Decode_NOP);
        op = SystemHintOp_TSB;
    when '0010 100'
        op = SystemHintOp_CSDB;
    when '0011 xxx'
        case op2 of
            when '000' SEE "PACIAZ";
            when '001' SEE "PACIASP";
            when '010' SEE "PACIBZ";
            when '011' SEE "PACIBSP";
            when '100' SEE "AUTIAZ";
            when '101' SEE "AUTIASP";
            when '110' SEE "AUTIBZ";
            when '111' SEE "AUTIBSP";
    otherwise EndOfDecode(Decode_NOP);
```

Assembler Symbols

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**imm**> Is a 7-bit unsigned immediate, in the range 0 to 127, encoded in the “CRm:op2” field.

The encodings that are allocated to architectural hint functionality are described in the ’Hints’ table in the ’Index by Encoding’.

Note

For allocated encodings of “CRm:op2”:

- A disassembler will disassemble the allocated instruction, rather than the `HINT` instruction.
- An assembler may support assembly of allocated encodings using `HINT` with the corresponding <imm> value, but it is not required to do so.

Operation

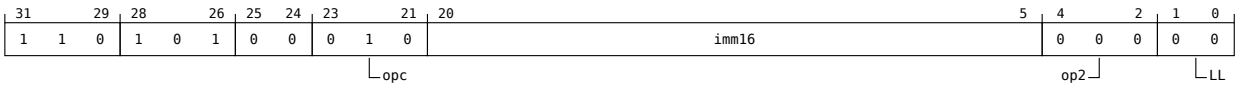
```
case op of
    when SystemHintOp_YIELD
        Hint_Yield();

    when SystemHintOp_DGH
        Hint_DGH();

    when SystemHintOp_WFE
        Hint_WFE();

    when SystemHintOp_WFI
        Hint_WFI();

    when SystemHintOp_SEV
        SendEvent();

    when SystemHintOp_SEVL
        SendEventLocal();

    when SystemHintOp_ESB
        SynchronizeErrors();
        AArch64.ESBOperation();
        if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
            AArch64.vESBOperation();
        TakeUnmaskedSErrorInterrupts();

    when SystemHintOp_TSB

        TraceSynchronizationBarrier();

    when SystemHintOp_CSDB
        ConsumptionOfSpeculativeDataBarrier();

    when SystemHintOp_NOP
        return; // Do nothing

    otherwise
        Unreachable();
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.89 HLT

Halt instruction

This instruction can generate a Halt Instruction debug event, which causes entry into Debug state.



Encoding

```
HLT #<imm>
```

Decode for this encoding

```
if EDSCR.HDE == '0' || !HaltingAllowed() then
    EndOfDecode(Decode_UNDEF);
```

Assembler Symbols

<imm> Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

Operation

```
constant boolean is_async = FALSE;
Halt(DebugHalt_HaltInstruction, is_async);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.90 HVC

Hypervisor call

This instruction causes an exception to EL2. Software executing at EL1 can use this instruction to call the hypervisor to request a service.

The HVC instruction is UNDEFINED:

- When HCR_EL2.HCD is set to 1.
- At EL1 if EL2 is not enabled.
- At EL0.

On executing an HVC instruction, the PE records the exception as a Hypervisor Call exception in ESR_ELx, using the EC syndrome value 0x16, and the value of the immediate argument.

Encoding

```
HVC #<imm>
```

Decode for this encoding

```
if !HaveEL(EL2) then EndOfDecode(Decode_UNDEF);
constant bits(16) imm = imm16;
```

Assembler Symbols

<imm> Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

Operation

```
if PSTATE.EL == EL0 then UNDEFINED;
if PSTATE.EL == EL1 && !EL2Enabled() then UNDEFINED;
if HCR_EL2.HCD == '1' then UNDEFINED;

AArch64.CallHypervisor(imm);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

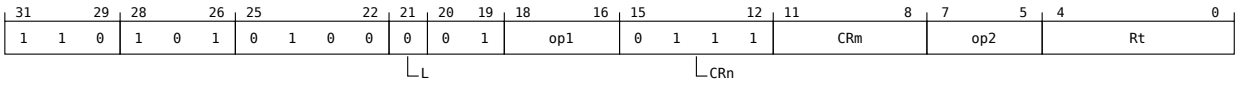
C6.2.91 IC

Instruction cache operation

For more information, see op0==0b01, cache maintenance, TLB maintenance, and address translation instructions.

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding

```
IC <ic_op>{, <Xt>}
```

is equivalent to

```
SYS #<op1>, C7, <Cm>, #<op2>{, <Xt>}
```

and is the preferred disassembly when `SysOp(op1, '0111', CRm, op2) == Sys_IC`.

Assembler Symbols

<ic_op> Is an IC operation name, as listed for the IC system instruction pages, encoded in "op1:CRm:op2":

op1	CRm	op2	<ic_op>
000	0001	000	IALLUIS
000	0101	000	IALLU
011	0101	001	IVAU

<Xt> Is the 64-bit name of the optional general-purpose source register, defaulting to '11111', encoded in the "Rt" field.

Operation

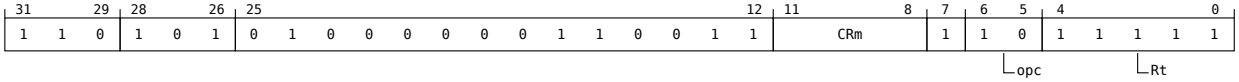
The description of SYS gives the operational pseudocode for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.92 ISB

Instruction synchronization barrier

This instruction flushes the pipeline in the PE and is a context synchronization event. For more information, see Instruction Synchronization Barrier (ISB).



Encoding

```
ISB {<option>|#<imm>}
```

Decode for this encoding

```
// no additional decoding required
```

Assembler Symbols

<**option**> Specifies an optional limitation on the barrier operation. Values are:

SY Full system barrier operation, encoded as CRm = 0b1111. Can be omitted.

All other encodings of "CRm" are reserved. The corresponding instructions execute as full system barrier operations, but must not be relied upon by software.

<**imm**> Is an optional 4-bit unsigned immediate, in the range 0 to 15, defaulting to 15 and encoded in the "CRm" field.

Operation

```
InstructionSynchronizationBarrier();
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.93 LDADD, LDADDA, LDADDAL, LDADDL

Atomic add on word or doubleword

This instruction atomically loads a 32-bit word or 64-bit doubleword from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

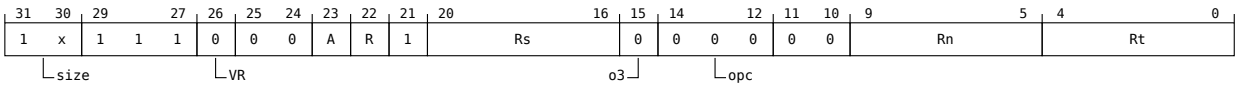
- If the destination register is not one of WZR or XZR, `LDADDA` and `LDADDAL` load from memory with acquire semantics.
- `LDADDL` and `LDADDAL` store to memory with release semantics.
- `LDADD` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STADD, STADDL.

Integer
(FEAT_LSE)



Encoding for the 32-bit no memory ordering variant Applies when `(size == 10 && A == 0 && R == 0)`

```
LDADD <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire variant Applies when `(size == 10 && A == 1 && R == 0)`

```
LDADDA <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire-release variant Applies when `(size == 10 && A == 1 && R == 1)`

```
LDADDAL <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit release variant Applies when `(size == 10 && A == 0 && R == 1)`

```
LDADDL <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 64-bit no memory ordering variant Applies when `(size == 11 && A == 0 && R == 0)`

```
LDADD <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit acquire variant Applies when `(size == 11 && A == 1 && R == 0)`

```
LDADDA <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit acquire-release variant Applies when `(size == 11 && A == 1 && R == 1)`

```
LDADDAL <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit release variant Applies when `(size == 11 && A == 0 && R == 1)`

```
LDADDL <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer datasize = 8 << UInt(size);
constant integer regsize = if datasize == 64 then 64 else 32;

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
```

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

Alias Conditions

Alias	Is preferred when
STADD, STADDL	A == '0' && Rt == '11111'

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_ADD, acquire,
                                                          release, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(datasize) comparevalue = bits(datasize) UNKNOWN; // Irrelevant when not executing CAS
constant bits(datasize) value = X[s, datasize];
constant bits(datasize) data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, regsize] = ZeroExtend(data, regsize);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.94 LDADDB, LDADDAB, LDADDALB, LDADDLB

Atomic add on byte

This instruction atomically loads an 8-bit byte from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

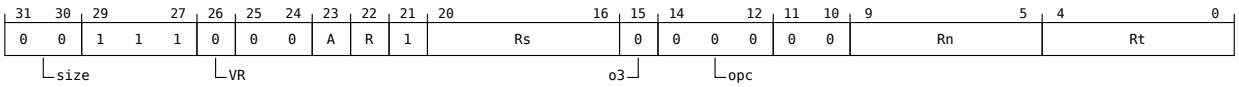
- If the destination register is not WZR, LDADDAB and LDADDALB load from memory with acquire semantics.
- LDADDLB and LDADDALB store to memory with release semantics.
- LDADDB has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STADDB, STADDLB.

Integer
(FEAT_LSE)



Encoding for the No memory ordering variant Applies when (A == 0 && R == 0)

```
LDADDB <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire variant Applies when (A == 1 && R == 0)

```
LDADDAB <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire-release variant Applies when (A == 1 && R == 1)

```
LDADDALB <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Release variant Applies when (A == 0 && R == 1)

```
LDADDLB <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
```

Assembler Symbols

<**Ws**> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<**Wt**> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias Conditions

Alias	Is preferred when
STADDB, STADDLB	A == '0' && Rt == '11111'

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_ADD, acquire,
                                                          release, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(8) comparevalue = bits(8) UNKNOWN; // Irrelevant when not executing CAS
constant bits(8) value = X[s, 8];
constant bits(8) data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.95 LDADDH, LDADDAH, LDADDALH, LDADDLH

Atomic add on halfword

This instruction atomically loads a 16-bit halfword from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

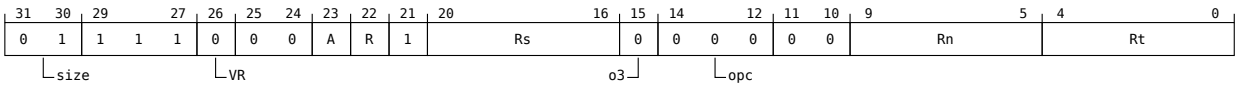
- If the destination register is not WZR, `LDADDAH` and `LDADDALH` load from memory with acquire semantics.
- `LDADDLH` and `LDADDALH` store to memory with release semantics.
- `LDADDH` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STADDH, STADDLH.

Integer
(FEAT_LSE)



Encoding for the No memory ordering variant Applies when `(A == 0 && R == 0)`

```
LDADDH <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire variant Applies when `(A == 1 && R == 0)`

```
LDADDAH <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire-release variant Applies when `(A == 1 && R == 1)`

```
LDADDALH <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Release variant Applies when `(A == 0 && R == 1)`

```
LDADDLH <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
```

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias Conditions

Alias	Is preferred when
STADDH, STADDLH	A == '0' && Rt == '11111'

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_ADD, acquire,
                                                          release, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(16) comparevalue = bits(16) UNKNOWN; // Irrelevant when not executing CAS
constant bits(16) value = X[s, 16];
constant bits(16) data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.96 LDAPR

Load-acquire RCpc register

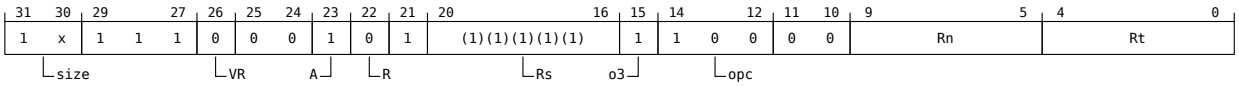
This instruction derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from the derived address in memory, and writes it to a register.

If the destination register is not one of WZR or XZR, `LDAPR` loads from memory with AcquirePC semantics.

For more information about memory ordering semantics, see Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.

No offset
(FEAT_LRCPC)



Encoding for the 32-bit variant Applies when `(size == 10)`

```
LDAPR <Wt>, [<Xn|SP> {, #0}]
```

Encoding for the 64-bit variant Applies when `(size == 11)`

```
LDAPR <Xt>, [<Xn|SP> {, #0}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LRCPC) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant boolean wback = FALSE;
constant integer offset = 0;
constant boolean wb_unknown = FALSE;
constant integer elsize = 8 << UInt(size);
constant integer regsize = if elsize == 64 then 64 else 32;
constant integer datasize = elsize;
constant boolean acquirepc = t != 31;
```

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

constant AccessDescriptor accdesc = CreateAccDescLDAcqPC(acquirepc, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
data = Mem[address, dbytes, accdesc];
X[t, regsize] = ZeroExtend(data, regsize);
if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    else
        address = AddressAdd(address, offset, accdesc);
    if n == 31 then
        SP[64] = address;
    else
        X[n, 64] = address;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.97 LDAPRB

Load-acquire RCpc register byte

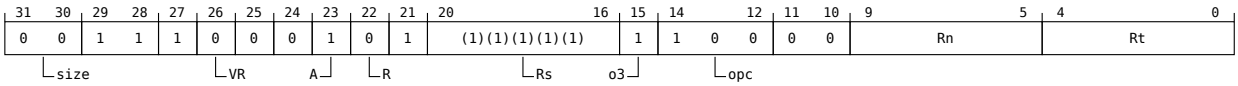
This instruction derives an address from a base register value, loads a byte from the derived address in memory, zero-extends it and writes it to a register.

If the destination register is not one of WZR or XZR, `LDAPRB` loads from memory with AcquirePC semantics.

For more information about memory ordering semantics, see Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Integer
(FEAT_LRCPC)



Encoding

```
LDAPRB <Wt>, [<Xn|SP> {, #0}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LRCPC) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant boolean acquirepc = t != 31;
```

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) data;

constant AccessDescriptor accdesc = CreateAccDescLDAcqPC(acquirepc, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

data = Mem[address, 1, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.98 LDAPRH

Load-acquire RCpc register halfword

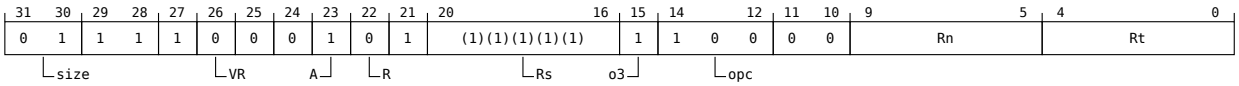
This instruction derives an address from a base register value, loads a halfword from the derived address in memory, zero-extends it and writes it to a register.

If the destination register is not one of WZR or XZR, `LDAPRH` loads from memory with AcquirePC semantics.

For more information about memory ordering semantics, see Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Integer
(FEAT_LRCPC)



Encoding

```
LDAPRH <Wt>, [<Xn|SP> {, #0}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LRCPC) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant boolean acquirepc = t != 31;
```

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) data;

constant AccessDescriptor accdesc = CreateAccDescLDAcqPC(acquirepc, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

data = Mem[address, 2, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.99 LDAPUR

Load-acquire RCpc register (unscaled)

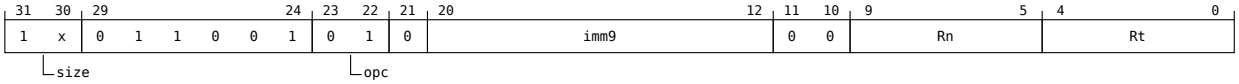
This instruction calculates an address from a base register and an immediate offset, loads a 32-bit word or 64-bit doubleword from memory, zero-extends it, and writes it to a register.

If the destination register is not one of WZR or XZR, `LDAPUR` loads from memory with AcquirePC semantics.

For more information about memory ordering semantics, see Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Unscaled offset
(FEAT_LRCPC2)



Encoding for the 32-bit variant Applies when `(size == 10)`

```
LDAPUR <Wt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 64-bit variant Applies when `(size == 11)`

```
LDAPUR <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LRCPC2) then EndOfDecode(Decode_UNDEF);
constant integer scale = UInt(size);
constant bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<**Wt**> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<**simm**> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

<**Xt**> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer datasize = 8 << scale;
constant integer regsize = if datasize == 64 then 64 else 32;
constant boolean acquirepc = t != 31;
```

Operation

```
bits(64) address;

constant AccessDescriptor accdesc = CreateAccDescLDAcqPC(acquirepc, t);

if n == 31 then
    CheckSPAlignment();
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

constant bits(datasize) data = Mem[address, datasize DIV 8, accdesc];
X[t, regsize] = ZeroExtend(data, regsize);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.100 LDAPURB

Load-acquire RCpc register byte (unscaled)

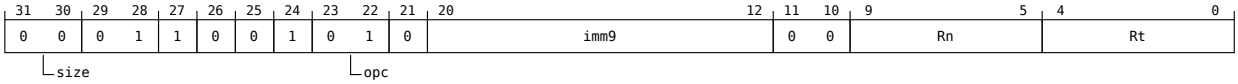
This instruction calculates an address from a base register and an immediate offset, loads a byte from memory, zero-extends it, and writes it to a register.

If the destination register is not one of WZR or XZR, `LDAPURB` loads from memory with AcquirePC semantics.

For more information about memory ordering semantics, see Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Unscaled offset
(FEAT_LRCPC2)



Encoding

```
LDAPURB <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LRCPC2) then EndOfDecode(Decode_UNDEF);
constant bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer datasize = 8;
constant boolean acquirepc = t != 31;
```

Operation

```
bits(64) address;

constant AccessDescriptor accdesc = CreateAccDescLDAcqPC(acquirepc, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

constant bits(datasize) data = Mem[address, datasize DIV 8, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.101 LDAPURH

Load-acquire RCpc register halfword (unscaled)

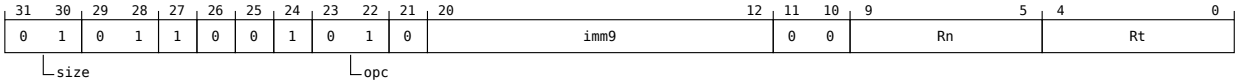
This instruction calculates an address from a base register and an immediate offset, loads a halfword from memory, zero-extends it, and writes it to a register.

If the destination register is not one of WZR or XZR, `LDAPURH` loads from memory with AcquirePC semantics.

For more information about memory ordering semantics, see Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Unscaled offset
(FEAT_LRCPC2)



Encoding

```
LDAPURH <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LRCPC2) then EndOfDecode(Decode_UNDEF);
constant bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<**Wt**> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<**simm**> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer datasize = 16;
constant boolean acquirepc = t != 31;
```

Operation

```
bits(64) address;

constant AccessDescriptor accdesc = CreateAccDescLDAcqPC(acquirepc, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

constant bits(datasize) data = Mem[address, datasize DIV 8, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.102 LDAPURSB

Load-acquire RCpc register signed byte (unscaled)

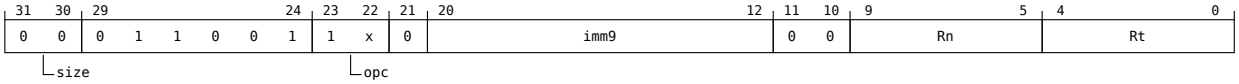
This instruction calculates an address from a base register and an immediate offset, loads a signed byte from memory, sign-extends it, and writes it to a register.

If the destination register is not one of WZR or XZR, `LDAPURSB` loads from memory with AcquirePC semantics.

For more information about memory ordering semantics, see Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Unscaled offset
(FEAT_LRCPC2)



Encoding for the 32-bit variant Applies when `(opc == 11)`

```
LDAPURSB <Wt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 64-bit variant Applies when `(opc == 10)`

```
LDAPURSB <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LRCPC2) then EndOfDecode(Decode_UNDEF);
constant bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<**Wt**> Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<**simm**> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

<**Xt**> Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer datasize = 8;
constant integer regsize = 64 >> UInt(opc<0>);
constant boolean acquirepc = t != 31;
```

Operation

```
bits(64) address;

constant AccessDescriptor accdesc = CreateAccDescLDAcqPC(acquirepc, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

constant bits(datasize) data = Mem[address, datasize DIV 8, accdesc];
X[t, regsize] = SignExtend(data, regsize);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.103 LDAPURSH

Load-acquire RCpc register signed halfword (unscaled)

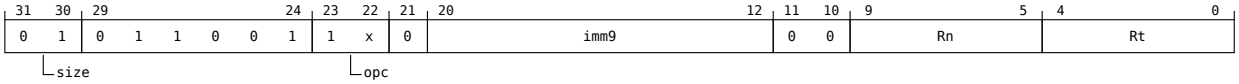
This instruction calculates an address from a base register and an immediate offset, loads a signed halfword from memory, sign-extends it, and writes it to a register.

If the destination register is not one of WZR or XZR, `LDAPURSH` loads from memory with AcquirePC semantics.

For more information about memory ordering semantics, see Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Unscaled offset
(FEAT_LRCPC2)



Encoding for the 32-bit variant Applies when `(opc == 11)`

```
LDAPURSH <Wt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 64-bit variant Applies when `(opc == 10)`

```
LDAPURSH <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LRCPC2) then EndOfDecode(Decode_UNDEF);
constant bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<**Wt**> Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<**simm**> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

<**Xt**> Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer datasize = 16;
constant integer regsize = 64 >> UInt(opc<0>);
constant boolean acquirepc = t != 31;
```

Operation

```
bits(64) address;

constant AccessDescriptor accdesc = CreateAccDescLDAcqPC(acquirepc, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

constant bits(datasize) data = Mem[address, datasize DIV 8, accdesc];
X[t, regsize] = SignExtend(data, regsize);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.104 LDAPURSW

Load-acquire RCpc register signed word (unscaled)

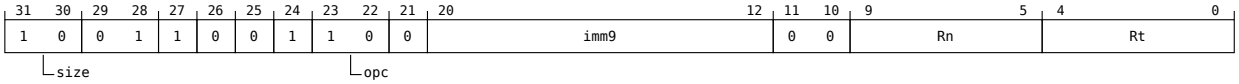
This instruction calculates an address from a base register and an immediate offset, loads a signed word from memory, sign-extends it, and writes it to a register.

If the destination register is not one of WZR or XZR, `LDAPURSW` loads from memory with AcquirePC semantics.

For more information about memory ordering semantics, see Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Unscaled offset
(FEAT_LRCPC2)



Encoding

```
LDAPURSW <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LRCPC2) then EndOfDecode(Decode_UNDEF);
constant bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer datasize = 32;
constant boolean acquirepc = t != 31;
```

Operation

```
bits(64) address;

constant AccessDescriptor accdesc = CreateAccDescLDAcqPC(acquirepc, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

constant bits(datasize) data = Mem[address, datasize DIV 8, accdesc];
X[t, 64] = SignExtend(data, 64);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.105 LDAR

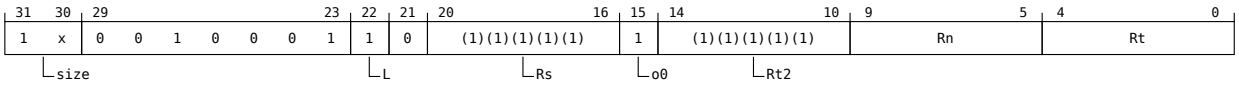
Load-acquire register

This instruction derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register.

If the destination register is not one of WZR or XZR, `LDAR` loads from memory with Acquire semantics.

For more information about memory ordering semantics, see Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.



Encoding for the 32-bit variant Applies when `(size == 10)`

```
LDAR <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit variant Applies when `(size == 11)`

```
LDAR <Xt>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer elsize = 8 << UInt(size);
constant integer regsize = if elsize == 64 then 64 else 32;
constant boolean acquire = t != 31;
```

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

Operation

```
bits(64) address;
constant integer dbytes = elsize DIV 8;

constant AccessDescriptor accdesc = CreateAccDescAcqRel(MemOp_LOAD, acquire, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(elsize) data = Mem[address,  dbytes, accdesc];
X[t, regsize] = ZeroExtend(data, regsize);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.106 LDARB

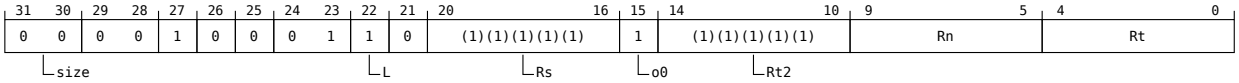
Load-acquire register byte

This instruction derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register.

If the destination register is not one of WZR or XZR, `LDARB` loads from memory with Acquire semantics.

For more information about memory ordering semantics, see Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.



Encoding

```
LDARB <Wt>, [<Xn|SP>{, #0}]
```

Decode for this encoding

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant boolean acquire = t != 31;
```

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;

constant AccessDescriptor accdesc = CreateAccDescAcqRel(MemOp_LOAD, acquire, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(8) data = Mem[address,  1, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.107 LDARH

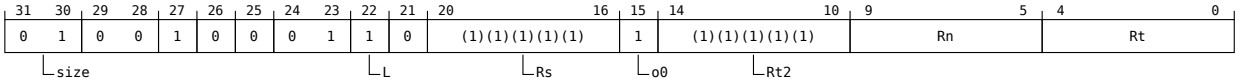
Load-acquire register halfword

This instruction derives an address from a base register value, loads a halfword from memory, zero-extends it, and writes it to a register.

If the destination register is not one of WZR or XZR, `LDARH` loads from memory with Acquire semantics.

For more information about memory ordering semantics, see Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.



Encoding

```
LDARH <Wt>, [<Xn|SP>{, #0}]
```

Decode for this encoding

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant boolean acquire = t != 31;
```

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;

constant AccessDescriptor accdesc = CreateAccDescAcqRel(MemOp_LOAD, acquire, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(16) data = Mem[address,  2, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.108 LDAXP

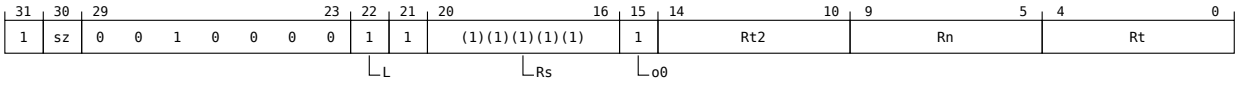
Load-acquire exclusive pair of registers

This instruction derives an address from a base register value, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. For information on single-copy atomicity and alignment requirements, see Requirements for single-copy atomicity and Alignment of data accesses. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores.

If the destination registers are not both WZR or not both XZR, `LDAXP` loads from memory with Acquire semantics.

For more information about memory ordering semantics, see Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.



Encoding for the 32-bit variant Applies when `(sz == 0)`

```
LDAXP <Wt1>, <Wt2>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit variant Applies when `(sz == 1)`

```
LDAXP <Xt1>, <Xt2>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
constant integer t = UInt(Rt);
constant integer t2 = UInt(Rt2);
constant integer n = UInt(Rn);

constant integer elsize = 32 << UInt(sz);
constant integer datasize = elsize * 2;
constant boolean acqrel = t != 31 && t2 != 31;

boolean rt_unknown = FALSE;
if t == t2 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDAXP.

Assembler Symbols

<**Wt1**> Is the 32-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<**Wt2**> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<**Xt1**> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<**Xt2**> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
bits(64) address;
bits(datasize) data;

constant integer dbytes = datasize DIV 8;
constant boolean privileged = PSTATE.EL != EL0;
constant boolean ispair = elsize == 64;   // When elsize is 32, the access is single-copy atomic
constant AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_LOAD, acqrel, privileged, ispair, t,
                                                        t2);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

AArch64.SetExclusiveMonitors(address, dbytes, accdesc);

data = Mem[address, dbytes, accdesc];
if rt_unknown then
    // ConstrainedUNPREDICTABLE case
    X[t, datasize]  = bits(datasize) UNKNOWN;        // In this case t = t2
elsif BigEndian(accdesc.acctype) then
    X[t, elsize]  = data<elsize +: elsize>;
    X[t2, elsize] = data<0 +: elsize>;
else
    X[t, elsize]  = data<0 +: elsize>;
    X[t2, elsize] = data<elsize +: elsize>;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.109 LDAXR

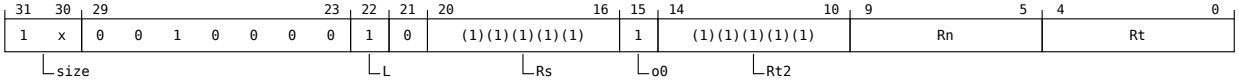
Load-acquire exclusive register

This instruction derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores.

If the destination register is not one of WZR or XZR, `LDAXR` loads from memory with Acquire semantics.

For more information about memory ordering semantics, see Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.



Encoding for the 32-bit variant Applies when `(size == 10)`

```
LDAXR <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit variant Applies when `(size == 11)`

```
LDAXR <Xt>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer elsize = 8 << UInt(size);
constant integer regsize = if elsize == 64 then 64 else 32;
constant boolean acqrel = t != 31;
```

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

Operation

```
bits(64) address;
bits(elsize) data;

constant integer dbytes = elsize DIV 8;
constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_LOAD, acqrel, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

AArch64.SetExclusiveMonitors(address, dbytes, accdesc);

data = Mem[address, dbytes, accdesc];
X[t, regsize] = ZeroExtend(data, regsize);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.110 LDAXRB

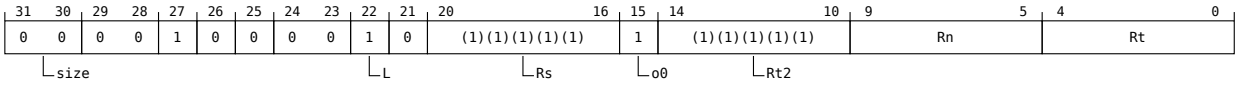
Load-acquire exclusive register byte

This instruction derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores.

If the destination register is not one of WZR or XZR, `LDAXRB` loads from memory with Acquire semantics.

For more information about memory ordering semantics, see Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.



Encoding

```
LDAXRB <Wt>, [<Xn|SP>{, #0}]
```

Decode for this encoding

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acqrel = t != 31;
```

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) data;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_LOAD, acqrel, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

AArch64.SetExclusiveMonitors(address, 1, accdesc);

data = Mem[address, 1, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.111 LDAXRH

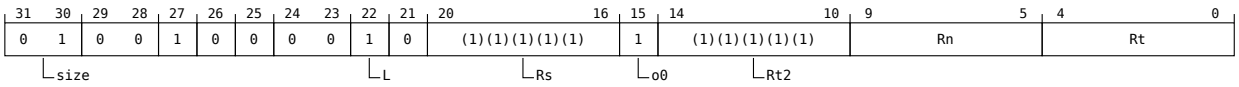
Load-acquire exclusive register halfword

This instruction derives an address from a base register value, loads a halfword from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores.

If the destination register is not one of WZR or XZR, `LDAXRH` loads from memory with Acquire semantics.

For more information about memory ordering semantics, see Load-Acquire, Load-AcquirePC, and Store-Release.

For information about addressing modes, see Load/Store addressing modes.



Encoding

```
LDAXRH <Wt>, [<Xn|SP>{, #0}]
```

Decode for this encoding

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acqrel = t != 31;
```

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) data;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_LOAD, acqrel, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

AArch64.SetExclusiveMonitors(address, 2, accdesc);

data = Mem[address, 2, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.112 LDCLR, LDCLRA, LDCLRAL, LDCLRL

Atomic bit clear on word or doubleword

This instruction atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

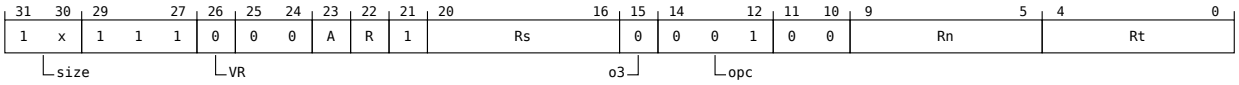
- If the destination register is not one of WZR or XZR, `LDCLRA` and `LDCLRAL` load from memory with acquire semantics.
- `LDCLRL` and `LDCLRAL` store to memory with release semantics.
- `LDCLR` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STCLR, STCLRL.

Integer
(FEAT_LSE)



Encoding for the 32-bit no memory ordering variant Applies when `(size == 10 && A == 0 && R == 0)`

```
LDCLR <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire variant Applies when `(size == 10 && A == 1 && R == 0)`

```
LDCLRA <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire-release variant Applies when `(size == 10 && A == 1 && R == 1)`

```
LDCLRAL <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit release variant Applies when `(size == 10 && A == 0 && R == 1)`

```
LDCLRL <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 64-bit no memory ordering variant Applies when `(size == 11 && A == 0 && R == 0)`

```
LDCLR <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit acquire variant Applies when `(size == 11 && A == 1 && R == 0)`

```
LDCLRA <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit acquire-release variant Applies when `(size == 11 && A == 1 && R == 1)`

```
LDCLRAL <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit release variant Applies when `(size == 11 && A == 0 && R == 1)`

```
LDCLRL <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer datasize = 8 << UInt(size);
constant integer regsize = if datasize == 64 then 64 else 32;

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
```

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

Alias Conditions

Alias	Is preferred when
STCLR, STCLRL	A == '0' && Rt == '11111'

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_BIC, acquire,
                                                          release, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(datasize) comparevalue = bits(datasize) UNKNOWN; // Irrelevant when not executing CAS
constant bits(datasize) value = X[s, datasize];
constant bits(datasize) data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, regsize] = ZeroExtend(data, regsize);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.113 LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB

Atomic bit clear on byte

This instruction atomically loads an 8-bit byte from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

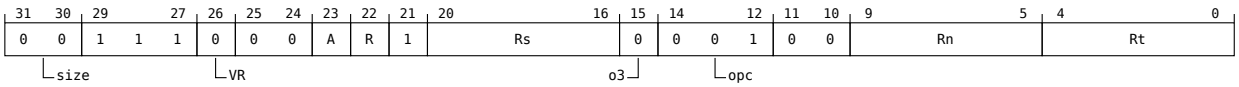
- If the destination register is not WZR, `LDCLRAB` and `LDCLRALB` load from memory with acquire semantics.
- `LDCLRLB` and `LDCLRALB` store to memory with release semantics.
- `LDCLRB` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STCLRB, STCLRLB.

Integer
(FEAT_LSE)



Encoding for the No memory ordering variant Applies when `(A == 0 && R == 0)`

```
LDCLRB <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire variant Applies when `(A == 1 && R == 0)`

```
LDCLRAB <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire-release variant Applies when `(A == 1 && R == 1)`

```
LDCLRALB <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Release variant Applies when `(A == 0 && R == 1)`

```
LDCLRLB <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
```

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias Conditions

Alias	Is preferred when
STCLRB, STCLRLB	A == '0' && Rt == '11111'

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_BIC, acquire,
                                                          release, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(8) comparevalue = bits(8) UNKNOWN; // Irrelevant when not executing CAS
constant bits(8) value = X[s, 8];
constant bits(8) data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.114 LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH

Atomic bit clear on halfword

This instruction atomically loads a 16-bit halfword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

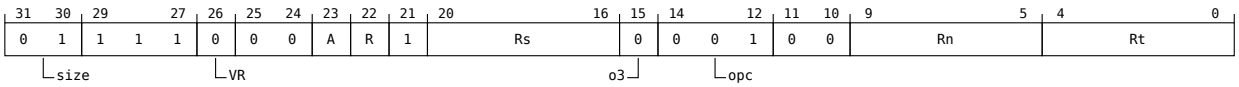
- If the destination register is not WZR, `LDCLRAH` and `LDCLRALH` load from memory with acquire semantics.
- `LDCLRLH` and `LDCLRALH` store to memory with release semantics.
- `LDCLRH` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STCLRH, STCLRLH.

Integer
(FEAT_LSE)



Encoding for the No memory ordering variant Applies when `(A == 0 && R == 0)`

```
LDCLRH <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire variant Applies when `(A == 1 && R == 0)`

```
LDCLRAH <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire-release variant Applies when `(A == 1 && R == 1)`

```
LDCLRALH <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Release variant Applies when `(A == 0 && R == 1)`

```
LDCLRLH <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
```

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias Conditions

Alias	Is preferred when
STCLRH, STCLRLH	A == '0' && Rt == '11111'

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_BIC, acquire,
                                                          release, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(16) comparevalue = bits(16) UNKNOWN; // Irrelevant when not executing CAS
constant bits(16) value = X[s, 16];
constant bits(16) data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.115 LDEOR, LDEORA, LDEORAL, LDEORL

Atomic exclusive-OR on word or doubleword

This instruction atomically loads a 32-bit word or 64-bit doubleword from memory, performs an exclusive-OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

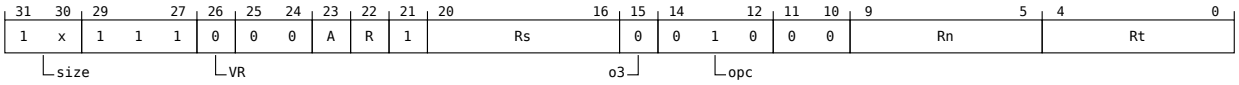
- If the destination register is not one of WZR or XZR, `LDEORA` and `LDEORAL` load from memory with acquire semantics.
- `LDEORL` and `LDEORAL` store to memory with release semantics.
- `LDEOR` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STEOR, STEORL.

Integer
(FEAT_LSE)



Encoding for the 32-bit no memory ordering variant Applies when `(size == 10 && A == 0 && R == 0)`

```
LDEOR <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire variant Applies when `(size == 10 && A == 1 && R == 0)`

```
LDEORA <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire-release variant Applies when `(size == 10 && A == 1 && R == 1)`

```
LDEORAL <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit release variant Applies when `(size == 10 && A == 0 && R == 1)`

```
LDEORL <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 64-bit no memory ordering variant Applies when `(size == 11 && A == 0 && R == 0)`

```
LDEOR <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit acquire variant Applies when `(size == 11 && A == 1 && R == 0)`

```
LDEORA <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit acquire-release variant Applies when `(size == 11 && A == 1 && R == 1)`

```
LDEORAL <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit release variant Applies when `(size == 11 && A == 0 && R == 1)`

```
LDEORL <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer datasize = 8 << UInt(size);
constant integer regsize = if datasize == 64 then 64 else 32;

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
```

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

Alias Conditions

Alias	Is preferred when
STEOR, STEORL	A == '0' && Rt == '11111'

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_EOR, acquire,
                                                          release, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(datasize) comparevalue = bits(datasize) UNKNOWN; // Irrelevant when not executing CAS
constant bits(datasize) value = X[s, datasize];
constant bits(datasize) data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, regsize] = ZeroExtend(data, regsize);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.116 LDEORB, LDEORAB, LDEORALB, LDEORLB

Atomic exclusive-OR on byte

This instruction atomically loads an 8-bit byte from memory, performs an exclusive-OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

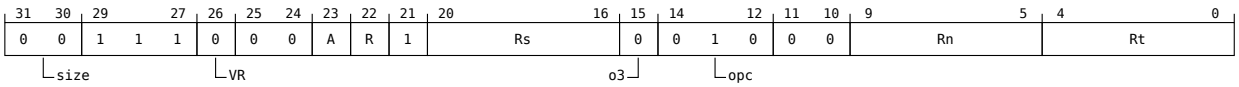
- If the destination register is not WZR, `LDEORAB` and `LDEORALB` load from memory with acquire semantics.
- `LDEORLB` and `LDEORALB` store to memory with release semantics.
- `LDEORB` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STEORB, STEORLB.

Integer
(FEAT_LSE)



Encoding for the No memory ordering variant Applies when `(A == 0 && R == 0)`

```
LDEORB <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire variant Applies when `(A == 1 && R == 0)`

```
LDEORAB <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire-release variant Applies when `(A == 1 && R == 1)`

```
LDEORALB <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Release variant Applies when `(A == 0 && R == 1)`

```
LDEORLB <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
```

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias Conditions

Alias	Is preferred when
STEORB, STEORLB	A == '0' && Rt == '11111'

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_EOR, acquire,
                                                          release, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(8) comparevalue = bits(8) UNKNOWN; // Irrelevant when not executing CAS
constant bits(8) value = X[s, 8];
constant bits(8) data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.117 LDEORH, LDEORAH, LDEORALH, LDEORLH

Atomic exclusive-OR on halfword

This instruction atomically loads a 16-bit halfword from memory, performs an exclusive-OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

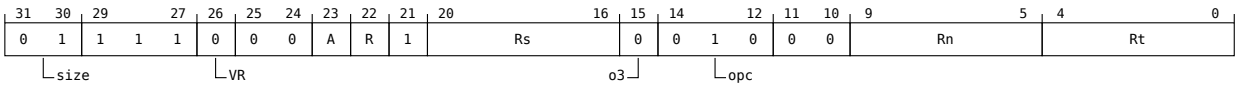
- If the destination register is not WZR, `LDEORAH` and `LDEORALH` load from memory with acquire semantics.
- `LDEORLH` and `LDEORALH` store to memory with release semantics.
- `LDEORH` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STEORH, STEORLH.

Integer
(FEAT_LSE)



Encoding for the No memory ordering variant Applies when `(A == 0 && R == 0)`

```
LDEORH <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire variant Applies when `(A == 1 && R == 0)`

```
LDEORAH <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire-release variant Applies when `(A == 1 && R == 1)`

```
LDEORALH <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Release variant Applies when `(A == 0 && R == 1)`

```
LDEORLH <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
```

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias Conditions

Alias	Is preferred when
STEORH, STEORLH	A == '0' && Rt == '11111'

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_EOR, acquire,
                                                          release, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(16) comparevalue = bits(16) UNKNOWN; // Irrelevant when not executing CAS
constant bits(16) value = X[s, 16];
constant bits(16) data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

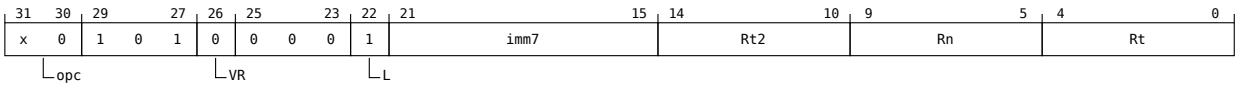
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.118 LDNP

Load pair of registers, with non-temporal hint

This instruction calculates an address from a base register value and an immediate offset, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers.

For information about addressing modes, see Load/Store addressing modes. For information about non-temporal pair instructions, see Load/Store non-temporal pair.



Encoding for the 32-bit variant Applies when `(opc == 00)`

```
LDNP <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]
```

Encoding for the 64-bit variant Applies when `(opc == 10)`

```
LDNP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

Decode for all variants of this encoding

```
constant integer t = UInt(Rt);
constant integer t2 = UInt(Rt2);
constant integer n = UInt(Rn);
constant boolean nontemporal = TRUE;
constant integer scale = 2 + UInt(opc<1>);
constant integer datasize = 8 << scale;
constant bits(64) offset = LSL(SignExtend(imm7, 64), scale);

boolean rt_unknown = FALSE;
if t == t2 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // Result is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDNP.

Assembler Symbols

<**Wt1**> Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<**Wt2**> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<**imm**> For the "32-bit" variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.

For the "64-bit" variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

<**Xt1**> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<**Xt2**> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
bits(64) address;
constant integer dbytes = datasize DIV 8;
constant boolean privileged = PSTATE.EL != EL0;
constant boolean ispair = TRUE;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged, ispair,
                                                     t, t2);
if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);
constant bits(2*datasize) data = Mem[address, 2 * dbytes, accdesc];
if rt_unknown then
    X[t, datasize]  = bits(datasize) UNKNOWN;
elsif BigEndian(accdesc.acctype) then
    X[t2, datasize] = data<(datasize-1):0>;
    X[t, datasize]  = data<(2*datasize-1):datasize>;
else
    X[t, datasize]  = data<(datasize-1):0>;
    X[t2, datasize] = data<(2*datasize-1):datasize>;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

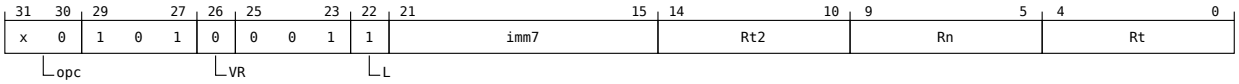
C6.2.119 LDP

Load pair of registers

This instruction calculates an address from a base register value and an immediate offset, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. For information about addressing modes, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index , Pre-index and Signed offset

Post-index



Encoding for the 32-bit variant Applies when `(opc == 00)`

```
LDP <Wt1>, <Wt2>, [<Xn|SP>], #<imm>
```

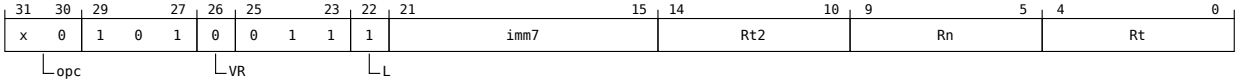
Encoding for the 64-bit variant Applies when `(opc == 10)`

```
LDP <Xt1>, <Xt2>, [<Xn|SP>], #<imm>
```

Decode for all variants of this encoding

```
boolean wback = TRUE;
constant boolean postindex = TRUE;
```

Pre-index



Encoding for the 32-bit variant Applies when `(opc == 00)`

```
LDP <Wt1>, <Wt2>, [<Xn|SP>, #<imm>]!
```

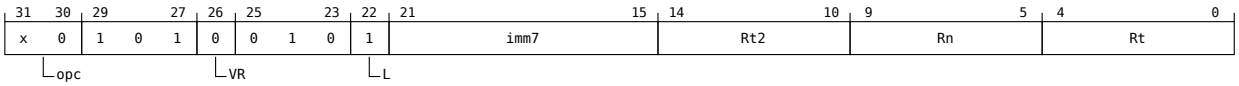
Encoding for the 64-bit variant Applies when `(opc == 10)`

```
LDP <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]!
```

Decode for all variants of this encoding

```
boolean wback = TRUE;
constant boolean postindex = FALSE;
```

Signed offset



Encoding for the 32-bit variant Applies when `(opc == 00)`

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
LDP <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]
```

Encoding for the 64-bit variant Applies when `(opc == 10)`

```
LDP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

Decode for all variants of this encoding

```
boolean wback = FALSE;
constant boolean postindex = FALSE;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDP.

Assembler Symbols

<**Wt1**> Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<**Wt2**> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<**imm**> For the "Post-index 32-bit" and "Pre-index 32-bit" variants: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4.

For the "Post-index 64-bit" and "Pre-index 64-bit" variants: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as <imm>/8.

For the "Signed offset 32-bit" variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.

For the "Signed offset 64-bit" variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

<**Xt1**> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<**Xt2**> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer t2 = UInt(Rt2);
constant integer n = UInt(Rn);
constant boolean nontemporal = FALSE;
constant integer scale = 2 + UInt(opc<1>);
constant integer datasize = 8 << scale;
constant bits(64) offset = LSL(SignExtend(imm7, 64), scale);

boolean rt_unknown = FALSE;
boolean wb_unknown = FALSE;

if wback && (t == n || t2 == n) && n != 31 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;        // Writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // Writeback is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);

if t == t2 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // Result is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

constant boolean privileged = PSTATE.EL != EL0;
constant boolean ispair = TRUE;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged, ispair,
                                                     t, t2);
if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

if !postindex then
    address = AddressAdd(address, offset, accdesc);

constant bits(2*datasize) data = Mem[address, 2 * dbytes, accdesc];
if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
    data2 = bits(datasize) UNKNOWN;
elsif BigEndian(accdesc.acctype) then
    data2 = data<(datasize-1):0>;
    data1 = data<(2*datasize-1):datasize>;
else
    data1 = data<(datasize-1):0>;
    data2 = data<(2*datasize-1):datasize>;

X[t,  datasize] = data1;
X[t2, datasize] = data2;

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = AddressAdd(address, offset, accdesc);
    if n == 31 then
        SP[64] = address;
    else
        X[n, 64] = address;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

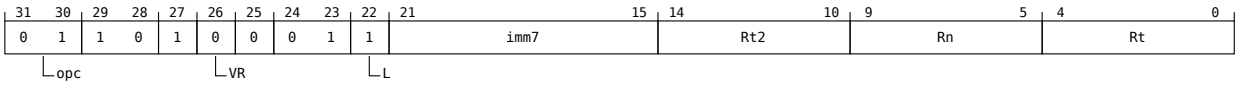
C6.2.120 LDPSW

Load pair of registers signed word

This instruction calculates an address from a base register value and an immediate offset, loads two 32-bit words from memory, sign-extends them, and writes them to two registers. For information about addressing modes, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index , Pre-index and Signed offset

Post-index



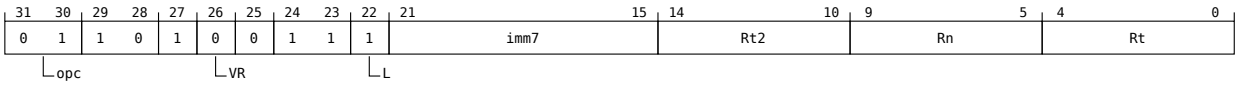
Encoding

```
LDPSW <Xt1>, <Xt2>, [<Xn|SP>], #<imm>
```

Decode for this encoding

```
boolean wback = TRUE;
constant boolean postindex = TRUE;
```

Pre-index



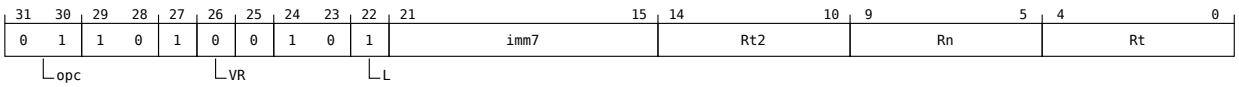
Encoding

```
LDPSW <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]!
```

Decode for this encoding

```
boolean wback = TRUE;
constant boolean postindex = FALSE;
```

Signed offset



Encoding

```
LDPSW <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

Decode for this encoding

```
boolean wback = FALSE;
constant boolean postindex = FALSE;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDPSW.

Assembler Symbols

<**Xt1**> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<**Xt2**> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<**imm**> For the "Post-index" and "Pre-index" variants: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4.

For the "Signed offset" variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer t2 = UInt(Rt2);
constant integer n = UInt(Rn);
constant boolean nontemporal = FALSE;
constant integer scale = 2 + UInt(opc<1>);
constant integer datasize = 32;
constant bits(64) offset = LSL(SignExtend(imm7, 64), scale);

boolean rt_unknown = FALSE;
boolean wb_unknown = FALSE;

if wback && (t == n || t2 == n) && n != 31 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;        // Writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // Writeback is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);

if t == t2 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // Result is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
```

Operation

```
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

constant boolean privileged = PSTATE.EL != EL0;
constant boolean ispair = TRUE;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged, ispair,
                                                     t, t2);
if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

if !postindex then
    address = AddressAdd(address, offset, accdesc);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant bits(2*datasize) data = Mem[address, 2 * dbytes, accdesc];
if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
    data2 = bits(datasize) UNKNOWN;
elsif BigEndian(accdesc.acctype) then
    data2 = data<(datasize-1):0>;
    data1 = data<(2*datasize-1):datasize>;
else
    data1 = data<(datasize-1):0>;
    data2 = data<(2*datasize-1):datasize>;

X[t,  64] = SignExtend(data1, 64);
X[t2, 64] = SignExtend(data2, 64);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = AddressAdd(address, offset, accdesc);
    if n == 31 then
        SP[64] = address;
    else
        X[n, 64] = address;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

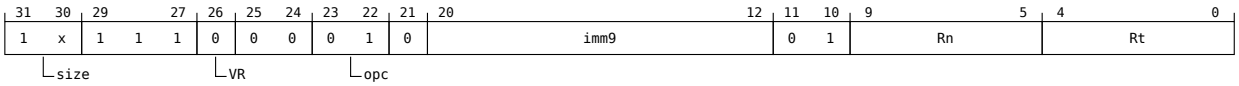
C6.2.121 LDR (immediate)

Load register (immediate)

This instruction loads a word or doubleword from memory and writes it to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about addressing modes, see Load/Store addressing modes. The Unsigned offset variant scales the immediate offset value by the size of the value accessed before adding it to the base register value.

It has encodings from 3 classes: Post-index , Pre-index and Unsigned offset

Post-index



Encoding for the 32-bit variant Applies when `(size == 10)`

```
LDR <Wt>, [<Xn|SP>], #<simm>
```

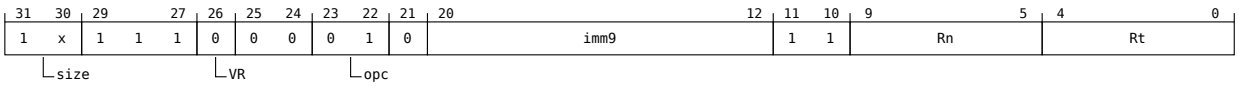
Encoding for the 64-bit variant Applies when `(size == 11)`

```
LDR <Xt>, [<Xn|SP>], #<simm>
```

Decode for all variants of this encoding

```
boolean wback = TRUE;
constant boolean postindex = TRUE;
constant integer scale = UInt(size);
constant bits(64) offset = SignExtend(imm9, 64);
```

Pre-index



Encoding for the 32-bit variant Applies when `(size == 10)`

```
LDR <Wt>, [<Xn|SP>, #<simm>]!
```

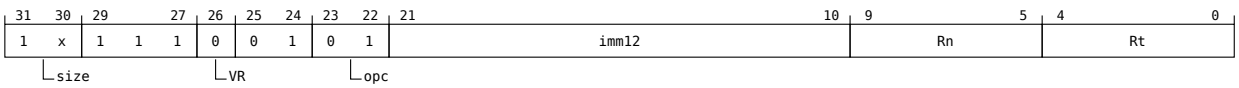
Encoding for the 64-bit variant Applies when `(size == 11)`

```
LDR <Xt>, [<Xn|SP>, #<simm>]!
```

Decode for all variants of this encoding

```
boolean wback = TRUE;
constant boolean postindex = FALSE;
constant integer scale = UInt(size);
constant bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset



Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 32-bit variant Applies when `(size == 10)`

```
LDR <Wt>, [<Xn|SP>{, #<pimm>}]
```

Encoding for the 64-bit variant Applies when `(size == 11)`

```
LDR <Xt>, [<Xn|SP>{, #<pimm>}]
```

Decode for all variants of this encoding

```
boolean wback = FALSE;
constant boolean postindex = FALSE;
constant integer scale = UInt(size);
constant bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDR (immediate).

Assembler Symbols

<**Wt**> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<**simm**> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.

<**Xt**> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<**pimm**> For the "32-bit" variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4.

For the "64-bit" variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer datasize = 8 << scale;
constant integer regsize = if datasize == 64 then 64 else 32;
constant boolean nontemporal = FALSE;

Constraint c;
boolean wb_unknown = FALSE;
if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // Writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // Writeback is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
```

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !postindex then
    address = AddressAdd(address, offset, accdesc);

constant bits(datasize) data = Mem[address, datasize DIV 8, accdesc];
X[t, regsize] = ZeroExtend(data, regsize);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = AddressAdd(address, offset, accdesc);
    if n == 31 then
        SP[64] = address;
    else
        X[n, 64] = address;
```

Operational information

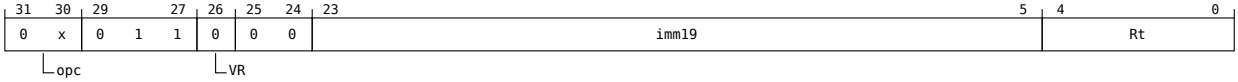
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.122 LDR (literal)

Load register (literal)

This instruction calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to a register. For information about addressing modes, see Load/Store addressing modes.



Encoding for the 32-bit variant Applies when `(opc == 00)`

```
LDR <Wt>, <label>
```

Encoding for the 64-bit variant Applies when `(opc == 01)`

```
LDR <Xt>, <label>
```

Decode for all variants of this encoding

```
constant integer t = UInt(Rt);
constant integer size = 4 << UInt(opc<0>);
constant boolean nontemporal = FALSE;
constant boolean tagchecked = FALSE;

constant bits(64) offset = SignExtend(imm19:'00', 64);
```

Assembler Symbols

<**Wt**> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<**label**> Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

<**Xt**> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

Operation

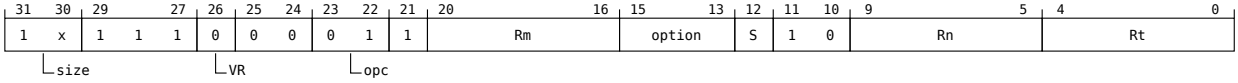
```
constant bits(64) address = PC64 + offset;
constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged, t);

X[t, size * 8] = Mem[address, size, accdesc];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.123 LDR (register)

Load register (register)

This instruction calculates an address from a base register value and an offset register value, loads a word from memory, and writes it to a register. The offset register value can optionally be shifted and extended. For information about addressing modes, see Load/Store addressing modes.



Encoding for the 32-bit variant Applies when `(size == 10)`

```
LDR <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Encoding for the 64-bit variant Applies when `(size == 11)`

```
LDR <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Decode for all variants of this encoding

```
if option<1> == '0' then EndOfDecode(Decode_UNDEF);             // sub-word index
constant ExtendType extend_type = DecodeRegExtend(option);
constant integer scale = UInt(size);
constant integer shift = if S == '1' then scale else 0;
```

Assembler Symbols

<**Wt**> Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<**Wm**> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the “Rm” field.

<**Xm**> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the “Rm” field.

<**extend**> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted, encoded in “option”:

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<**amount**> For the “32-bit” variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

S	<amount>
1	#2

For the “64-bit” variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#3

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 8 << scale;
constant integer regsize = if datasize == 64 then 64 else 32;
constant boolean nontemporal = FALSE;
```

Operation

```
constant bits(64) offset = ExtendReg(m, extend_type, shift, 64);
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

constant bits(datasize) data = Mem[address, datasize DIV 8, accdesc];
X[t, regsize] = ZeroExtend(data, regsize);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.124 LDRAA, LDRAB

Load register, with pointer authentication

This instruction authenticates an address from a base register using a modifier of zero and the specified key, adds an immediate offset to the authenticated address, and loads a 64-bit doubleword from memory at this resulting address into a register.

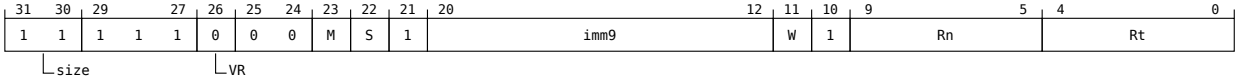
Key A is used for `LDRAA`. Key B is used for `LDRAB`.

If the authentication passes, the PE behaves the same as for an `LDR` instruction. For information on behavior if the authentication fails, see Faulting on pointer authentication.

The authenticated address is not written back to the base register, unless the pre-indexed variant of the instruction is used. In this case, the address that is written back to the base register does not include the pointer authentication code.

For information about addressing modes, see Load/Store addressing modes.

PAC
(FEAT_PAuth)



Encoding for the Key A, offset variant Applies when `(M == 0 && W == 0)`

```
LDRAA <Xt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the Key A, pre-indexed variant Applies when `(M == 0 && W == 1)`

```
LDRAA <Xt>, [<Xn|SP>{, #<simm>}]!
```

Encoding for the Key B, offset variant Applies when `(M == 1 && W == 0)`

```
LDRAB <Xt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the Key B, pre-indexed variant Applies when `(M == 1 && W == 1)`

```
LDRAB <Xt>, [<Xn|SP>{, #<simm>}]!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
boolean wback = W == '1';
constant boolean use_key_a = M == '0';
constant bits(10) S10 = S:imm9;
constant bits(64) offset = LSL(SignExtend(S10, 64), 3);
constant boolean nontemporal = FALSE;

boolean wb_unknown = FALSE;
if wback && n == t && n != 31 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<**Xt**> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<**simm**> Is the optional signed immediate byte offset, a multiple of 8 in the range -4096 to 4088, defaulting to 0 and encoded in the "S:imm9" field as <simm>/8.

Operation

```
bits(64) address;
constant boolean privileged = PSTATE.EL != EL0;
constant boolean auth_then_branch = TRUE;

constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged, t);
if n == 31 then
    address = SP[64];
else
    address = X[n, 64];

if use_key_a then
    address = AuthDA(address, X[31, 64], auth_then_branch);
else
    address = AuthDB(address, X[31, 64], auth_then_branch);

if n == 31 then
    CheckSPAlignment();

address = AddressAdd(address, offset, accdesc);
X[t, 64] = Mem[address, 8, accdesc];

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    if n == 31 then
        SP[64] = address;
    else
        X[n, 64] = address;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

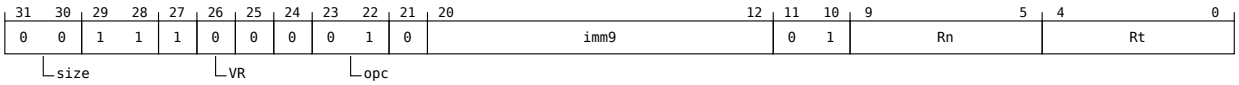
C6.2.125 LDRB (immediate)

Load register byte (immediate)

This instruction loads a byte from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about addressing modes, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index , Pre-index and Unsigned offset

Post-index



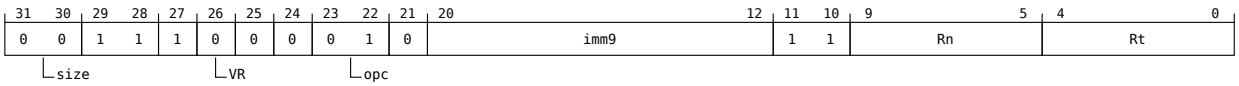
Encoding

```
LDRB <Wt>, [<Xn|SP>], #<simm>
```

Decode for this encoding

```
boolean wback = TRUE;
constant boolean postindex = TRUE;
constant bits(64) offset = SignExtend(imm9, 64);
```

Pre-index



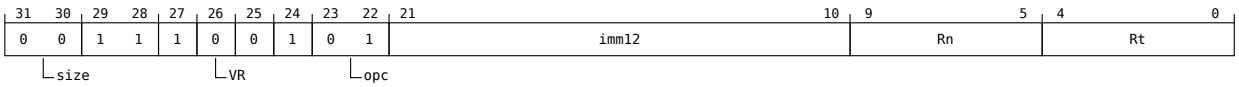
Encoding

```
LDRB <Wt>, [<Xn|SP>, #<simm>]!
```

Decode for this encoding

```
boolean wback = TRUE;
constant boolean postindex = FALSE;
constant bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset



Encoding

```
LDRB <Wt>, [<Xn|SP>{, #<pimm>}]
```

Decode for this encoding

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
boolean wback = FALSE;
constant boolean postindex = FALSE;
constant bits(64) offset = LSL(ZeroExtend(imm12, 64), 0);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDRH (immediate).

Assembler Symbols

<**Wt**> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<**simm**> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.

<**pimm**> Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant boolean nontemporal = FALSE;

Constraint c;
boolean wb_unknown = FALSE;
if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // Writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // Writeback is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
```

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

if !postindex then
    address = AddressAdd(address, offset, accdesc);

constant bits(8) data = Mem[address, 1, accdesc];
X[t, 32] = ZeroExtend(data, 32);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = AddressAdd(address, offset, accdesc);
    if n == 31 then
        SP[64] = address;
    else
        X[n, 64] = address;
```

Operational information

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

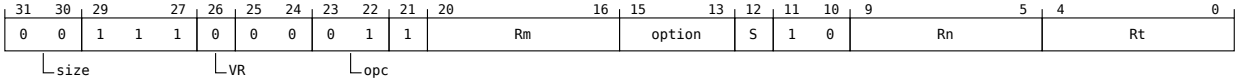
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.126 LDRB (register)

Load register byte (register)

This instruction calculates an address from a base register value and an offset register value, loads a byte from memory, zero-extends it, and writes it to a register. For information about addressing modes, see Load/Store addressing modes.



Encoding for the Extended register variant Applies when `(option != 011)`

```
LDRB <Wt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

Encoding for the Shifted register variant Applies when `(option == 011)`

```
LDRB <Wt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

Decode for all variants of this encoding

```
if option<1> == '0' then EndOfDecode(Decode_UNDEF);             // sub-word index
constant ExtendType extend_type = DecodeRegExtend(option);
constant integer shift = 0;
```

Assembler Symbols

<**Wt**> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<**Wm**> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

<**Xm**> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<**extend**> Is the index extend specifier, encoded in "option":

option	<extend>
010	UXTW
110	SXTW
111	SXTX

<**amount**> Is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant boolean nontemporal = FALSE;
```

Operation

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant bits(64) offset = ExtendReg(m, extend_type, shift, 64);
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

constant bits(8) data = Mem[address, 1, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

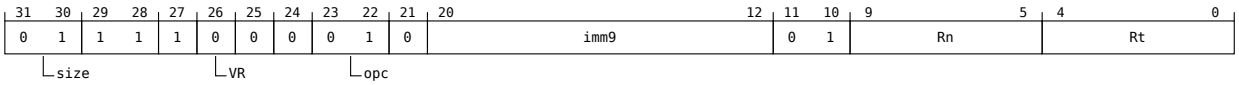
C6.2.127 LDRH (immediate)

Load register halfword (immediate)

This instruction loads a halfword from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about addressing modes, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index , Pre-index and Unsigned offset

Post-index



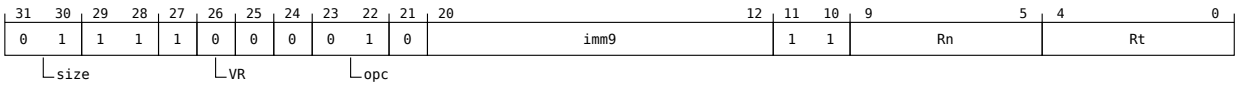
Encoding

```
LDRH <Wt>, [<Xn|SP>], #<simm>
```

Decode for this encoding

```
boolean wback = TRUE;
constant boolean postindex = TRUE;
constant bits(64) offset = SignExtend(imm9, 64);
```

Pre-index



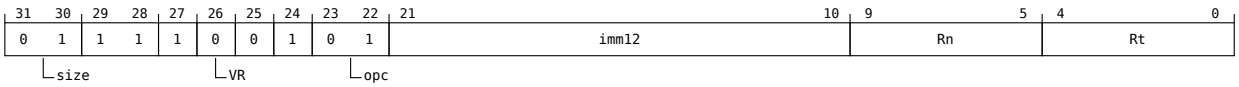
Encoding

```
LDRH <Wt>, [<Xn|SP>, #<simm>]!
```

Decode for this encoding

```
boolean wback = TRUE;
constant boolean postindex = FALSE;
constant bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset



Encoding

```
LDRH <Wt>, [<Xn|SP>{, #<pimm>}]
```

Decode for this encoding

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
boolean wback = FALSE;
constant boolean postindex = FALSE;
constant bits(64) offset = LSL(ZeroExtend(imm12, 64), 1);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDRH (immediate).

Assembler Symbols

<**Wt**> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<**simm**> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.

<**pimm**> Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant boolean nontemporal = FALSE;

Constraint c;
boolean wb_unknown = FALSE;
if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // Writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // Writeback is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
```

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

if !postindex then
    address = AddressAdd(address, offset, accdesc);

constant bits(16) data = Mem[address, 2, accdesc];
X[t, 32] = ZeroExtend(data, 32);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = AddressAdd(address, offset, accdesc);
    if n == 31 then
        SP[64] = address;
    else
        X[n, 64] = address;
```

Operational information

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

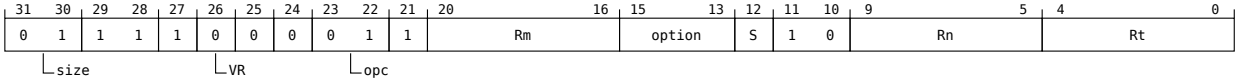
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.128 LDRH (register)

Load register halfword (register)

This instruction calculates an address from a base register value and an offset register value, loads a halfword from memory, zero-extends it, and writes it to a register. For information about addressing modes, see Load/Store addressing modes.



Encoding

```
LDRH <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Decode for this encoding

```
if option<1> == '0' then EndOfDecode(Decode_UNDEF);             // sub-word index
constant ExtendType extend_type = DecodeRegExtend(option);
constant integer shift = if S == '1' then 1 else 0;
```

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted, encoded in "option":

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<amount> Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

S	<amount>
0	#0
1	#1

Shared Decode

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant boolean nontemporal = FALSE;
```

Operation

```
constant bits(64) offset = ExtendReg(m, extend_type, shift, 64);
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

constant bits(16) data = Mem[address, 2, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

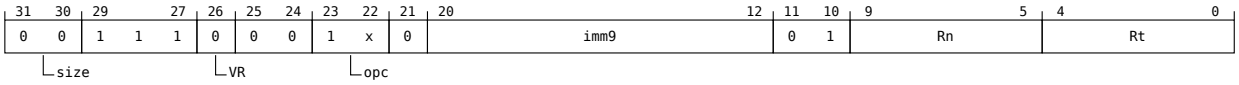
C6.2.129 LDRSB (immediate)

Load register signed byte (immediate)

This instruction loads a byte from memory, sign-extends it to either 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about addressing modes, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index , Pre-index and Unsigned offset

Post-index



Encoding for the 32-bit variant Applies when `(opc == 11)`

```
LDRSB <Wt>, [<Xn|SP>], #<simm>
```

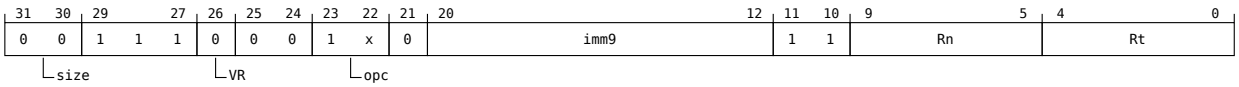
Encoding for the 64-bit variant Applies when `(opc == 10)`

```
LDRSB <Xt>, [<Xn|SP>], #<simm>
```

Decode for all variants of this encoding

```
boolean wback = TRUE;
constant boolean postindex = TRUE;
constant bits(64) offset = SignExtend(imm9, 64);
```

Pre-index



Encoding for the 32-bit variant Applies when `(opc == 11)`

```
LDRSB <Wt>, [<Xn|SP>, #<simm>]!
```

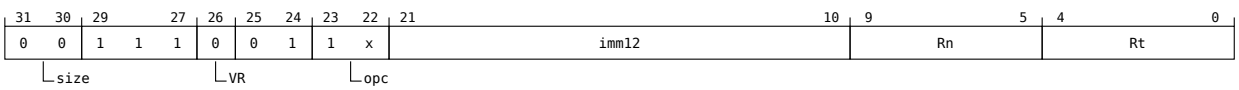
Encoding for the 64-bit variant Applies when `(opc == 10)`

```
LDRSB <Xt>, [<Xn|SP>, #<simm>]!
```

Decode for all variants of this encoding

```
boolean wback = TRUE;
constant boolean postindex = FALSE;
constant bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset



Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 32-bit variant Applies when `(opc == 11)`

```
LDRSB <Wt>, [<Xn|SP>{, #<pimm>}]
```

Encoding for the 64-bit variant Applies when `(opc == 10)`

```
LDRSB <Xt>, [<Xn|SP>{, #<pimm>}]
```

Decode for all variants of this encoding

```
boolean wback = FALSE;
constant boolean postindex = FALSE;
constant bits(64) offset = LSL(ZeroExtend(imm12, 64), 0);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDRSB (immediate).

Assembler Symbols

<**Wt**> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<**simm**> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.

<**Xt**> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<**pimm**> Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer regsize = 64 >> UInt(opc<0>);
constant boolean nontemporal = FALSE;

Constraint c;
boolean wb_unknown = FALSE;
if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // Writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // Writeback is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
```

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

if !postindex then
    address = AddressAdd(address, offset, accdesc);

constant bits(8) data = Mem[address, 1, accdesc];
X[t, regsize] = SignExtend(data, regsize);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = AddressAdd(address, offset, accdesc);
    if n == 31 then
        SP[64] = address;
    else
        X[n, 64] = address;
```

Operational information

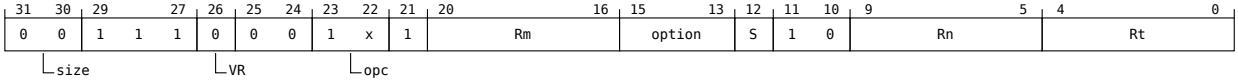
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.130 LDRSB (register)

Load register signed byte (register)

This instruction calculates an address from a base register value and an offset register value, loads a byte from memory, sign-extends it, and writes it to a register. For information about addressing modes, see Load/Store addressing modes.



Encoding for the 32-bit with extended register offset variant Applies when `(opc == 11 && option != 011)`

```
LDRSB <Wt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

Encoding for the 32-bit with shifted register offset variant Applies when `(opc == 11 && option == 011)`

```
LDRSB <Wt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

Encoding for the 64-bit with extended register offset variant Applies when `(opc == 10 && option != 011)`

```
LDRSB <Xt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

Encoding for the 64-bit with shifted register offset variant Applies when `(opc == 10 && option == 011)`

```
LDRSB <Xt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

Decode for all variants of this encoding

```
if option<1> == '0' then EndOfDecode(Decode_UNDEF);             // sub-word index
constant ExtendType extend_type = DecodeRegExtend(option);
constant integer shift = 0;
```

Assembler Symbols

<**Wt**> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<**Wm**> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

<**Xm**> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<**extend**> Is the index extend specifier, encoded in "option":

option	<extend>
010	UXTW
110	SXTW
111	SXTX

<**amount**> Is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.

<**Xt**> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer regsize = 64 >> UInt(opc<0>);
constant boolean nontemporal = FALSE;
```

Operation

```
constant bits(64) offset = ExtendReg(m, extend_type, shift, 64);
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

constant bits(8) data = Mem[address, 1, accdesc];
X[t, regsize] = SignExtend(data, regsize);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

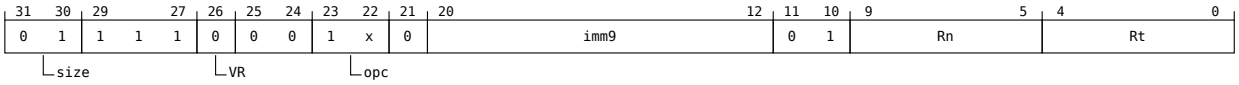
C6.2.131 LDRSH (immediate)

Load register signed halfword (immediate)

This instruction loads a halfword from memory, sign-extends it to 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about addressing modes, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index , Pre-index and Unsigned offset

Post-index



Encoding for the 32-bit variant Applies when `(opc == 11)`

```
LDRSH <Wt>, [<Xn|SP>], #<simm>
```

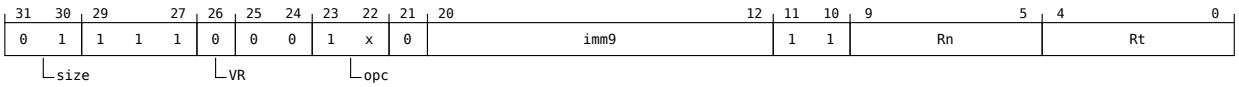
Encoding for the 64-bit variant Applies when `(opc == 10)`

```
LDRSH <Xt>, [<Xn|SP>], #<simm>
```

Decode for all variants of this encoding

```
boolean wback = TRUE;
constant boolean postindex = TRUE;
constant bits(64) offset = SignExtend(imm9, 64);
```

Pre-index



Encoding for the 32-bit variant Applies when `(opc == 11)`

```
LDRSH <Wt>, [<Xn|SP>, #<simm>]!
```

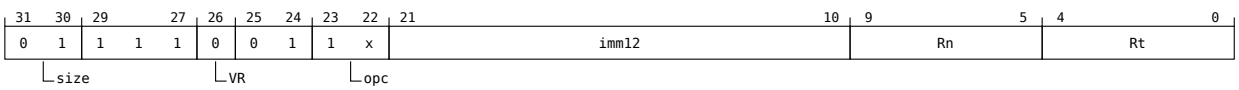
Encoding for the 64-bit variant Applies when `(opc == 10)`

```
LDRSH <Xt>, [<Xn|SP>, #<simm>]!
```

Decode for all variants of this encoding

```
boolean wback = TRUE;
constant boolean postindex = FALSE;
constant bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset



Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 32-bit variant Applies when `(opc == 11)`

```
LDRSH <Wt>, [<Xn|SP>{, #<pimm>}]
```

Encoding for the 64-bit variant Applies when `(opc == 10)`

```
LDRSH <Xt>, [<Xn|SP>{, #<pimm>}]
```

Decode for all variants of this encoding

```
boolean wback = FALSE;
constant boolean postindex = FALSE;
constant bits(64) offset = LSL(ZeroExtend(imm12, 64), 1);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDRSH (immediate).

Assembler Symbols

<**Wt**> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<**simm**> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.

<**Xt**> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<**pimm**> Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer regsize = 64 >> UInt(opc<0>);
constant boolean nontemporal = FALSE;

Constraint c;
boolean wb_unknown = FALSE;
if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // Writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // Writeback is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
```

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

if !postindex then
    address = AddressAdd(address, offset, accdesc);

constant bits(16) data = Mem[address, 2, accdesc];
X[t, regsize] = SignExtend(data, regsize);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = AddressAdd(address, offset, accdesc);
    if n == 31 then
        SP[64] = address;
    else
        X[n, 64] = address;
```

Operational information

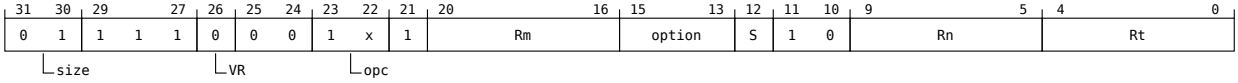
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.132 LDRSH (register)

Load register signed halfword (register)

This instruction calculates an address from a base register value and an offset register value, loads a halfword from memory, sign-extends it, and writes it to a register. For information about addressing modes, see Load/Store addressing modes.



Encoding for the 32-bit variant Applies when `(opc == 11)`

```
LDRSH <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Encoding for the 64-bit variant Applies when `(opc == 10)`

```
LDRSH <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Decode for all variants of this encoding

```
if option<1> == '0' then EndOfDecode(Decode_UNDEF);             // sub-word index
constant ExtendType extend_type = DecodeRegExtend(option);
constant integer shift = if S == '1' then 1 else 0;
```

Assembler Symbols

<**Wt**> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<**Wm**> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

<**Xm**> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<**extend**> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted, encoded in "option":

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<**amount**> Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

S	<amount>
0	#0
1	#1

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**Xt**> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer regsize = 64 >> UInt(opc<0>);
constant boolean nontemporal = FALSE;
```

Operation

```
constant bits(64) offset = ExtendReg(m, extend_type, shift, 64);
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

constant bits(16) data = Mem[address, 2, accdesc];
X[t, regsize] = SignExtend(data, regsize);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

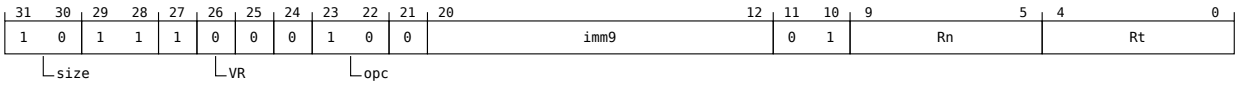
C6.2.133 LDRSW (immediate)

Load register signed word (immediate)

This instruction loads a word from memory, sign-extends it to 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about addressing modes, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index , Pre-index and Unsigned offset

Post-index



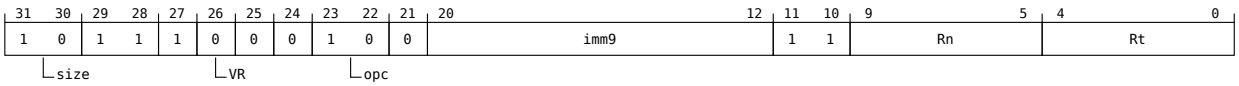
Encoding

```
LDRSW <Xt>, [<Xn|SP>], #<simm>
```

Decode for this encoding

```
boolean wback = TRUE;
constant boolean postindex = TRUE;
constant bits(64) offset = SignExtend(imm9, 64);
```

Pre-index



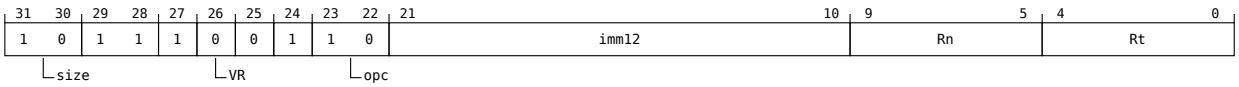
Encoding

```
LDRSW <Xt>, [<Xn|SP>, #<simm>]!
```

Decode for this encoding

```
boolean wback = TRUE;
constant boolean postindex = FALSE;
constant bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset



Encoding

```
LDRSW <Xt>, [<Xn|SP>{, #<pimm>}]
```

Decode for this encoding

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
boolean wback = FALSE;
constant boolean postindex = FALSE;
constant bits(64) offset = LSL(ZeroExtend(imm12, 64), 2);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDRSW (immediate).

Assembler Symbols

<**Xt**> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<**simm**> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.

<**pimm**> Is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant boolean nontemporal = FALSE;

Constraint c;
boolean wb_unknown = FALSE;
if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // Writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // Writeback is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
```

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

if !postindex then
    address = AddressAdd(address, offset, accdesc);

constant bits(32) data = Mem[address, 4, accdesc];
X[t, 64] = SignExtend(data, 64);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = AddressAdd(address, offset, accdesc);
    if n == 31 then
        SP[64] = address;
    else
        X[n, 64] = address;
```

Operational information

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

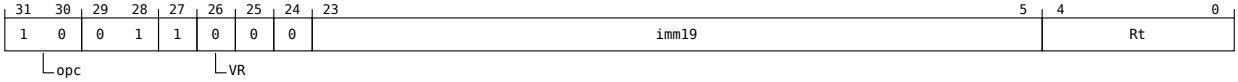
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.134 LDRSW (literal)

Load register signed word (literal)

This instruction calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to a register. For information about addressing modes, see Load/Store addressing modes.



Encoding

```
LDRSW <Xt>, <label>
```

Decode for this encoding

```
constant integer t = UInt(Rt);
constant boolean nontemporal = FALSE;
constant boolean tagchecked = FALSE;

constant bits(64) offset = SignExtend(imm19:'00', 64);
```

Assembler Symbols

<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<label> Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

Operation

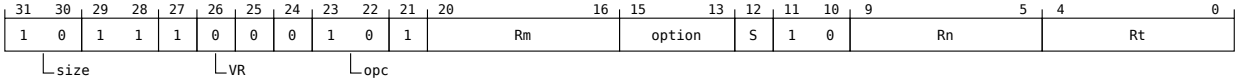
```
constant bits(64) address = PC64 + offset;
constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged, t);

constant bits(32) data = Mem[address, 4, accdesc];
X[t, 64] = SignExtend(data, 64);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.135 LDRSW (register)

Load register signed word (register)

This instruction calculates an address from a base register value and an offset register value, loads a word from memory, sign-extends it to form a 64-bit value, and writes it to a register. The offset register value can be shifted left by 0 or 2 bits. For information about addressing modes, see Load/Store addressing modes.



Encoding

```
LDRSW <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Decode for this encoding

```
if option<1> == '0' then EndOfDecode(Decode_UNDEF);             // sub-word index
constant ExtendType extend_type = DecodeRegExtend(option);
constant integer shift = if S == '1' then 2 else 0;
```

Assembler Symbols

<**Xt**> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<**Wm**> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

<**Xm**> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<**extend**> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted, encoded in "option":

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<**amount**> Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

S	<amount>
0	#0
1	#2

Shared Decode

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant boolean nontemporal = FALSE;
```

Operation

```
constant bits(64) offset = ExtendReg(m, extend_type, shift, 64);
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

constant bits(32) data = Mem[address, 4, accdesc];
X[t, 64] = SignExtend(data, 64);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.136 LDSET, LDSETA, LDSETAL, LDSETL

Atomic bit set on word or doubleword

This instruction atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

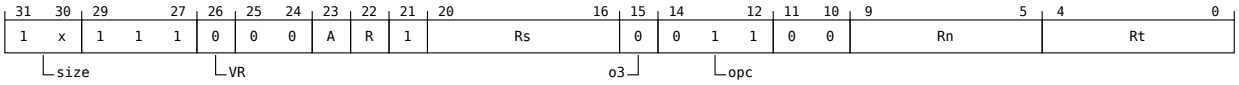
- If the destination register is not one of WZR or XZR, `LDSETA` and `LDSETAL` load from memory with acquire semantics.
- `LDSETL` and `LDSETAL` store to memory with release semantics.
- `LDSET` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STSET, STSETL.

Integer
(FEAT_LSE)



Encoding for the 32-bit no memory ordering variant Applies when `(size == 10 && A == 0 && R == 0)`

```
LDSET <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire variant Applies when `(size == 10 && A == 1 && R == 0)`

```
LDSETA <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire-release variant Applies when `(size == 10 && A == 1 && R == 1)`

```
LDSETAL <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit release variant Applies when `(size == 10 && A == 0 && R == 1)`

```
LDSETL <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 64-bit no memory ordering variant Applies when `(size == 11 && A == 0 && R == 0)`

```
LDSET <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit acquire variant Applies when `(size == 11 && A == 1 && R == 0)`

```
LDSETA <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit acquire-release variant Applies when `(size == 11 && A == 1 && R == 1)`

```
LDSETAL <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit release variant Applies when `(size == 11 && A == 0 && R == 1)`

```
LDSETL <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer datasize = 8 << UInt(size);
constant integer regsize = if datasize == 64 then 64 else 32;

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
```

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

Alias Conditions

Alias	Is preferred when
STSET, STSETL	A == '0' && Rt == '11111'

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_ORR, acquire,
                                                          release, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(datasize) comparevalue = bits(datasize) UNKNOWN; // Irrelevant when not executing CAS
constant bits(datasize) value = X[s, datasize];
constant bits(datasize) data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, regsize] = ZeroExtend(data, regsize);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.137 LDSETB, LDSETAB, LDSETALB, LDSETLB

Atomic bit set on byte

This instruction atomically loads an 8-bit byte from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

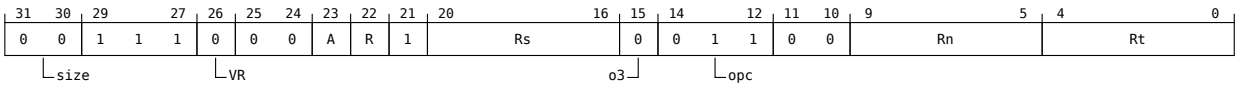
- If the destination register is not WZR, `LDSETAB` and `LDSETALB` load from memory with acquire semantics.
- `LDSETLB` and `LDSETALB` store to memory with release semantics.
- `LDSETB` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STSETB, STSETLB.

Integer
(FEAT_LSE)



Encoding for the No memory ordering variant Applies when `(A == 0 && R == 0)`

```
LDSETB <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire variant Applies when `(A == 1 && R == 0)`

```
LDSETAB <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire-release variant Applies when `(A == 1 && R == 1)`

```
LDSETALB <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Release variant Applies when `(A == 0 && R == 1)`

```
LDSETLB <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
```

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias Conditions

Alias	Is preferred when
STSETB, STSETLB	A == '0' && Rt == '11111'

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_ORR, acquire,
                                                          release, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(8) comparevalue = bits(8) UNKNOWN; // Irrelevant when not executing CAS
constant bits(8) value = X[s, 8];
constant bits(8) data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.138 LDSETH, LDSETAH, LDSETALH, LDSETLH

Atomic bit set on halfword

This instruction atomically loads a 16-bit halfword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

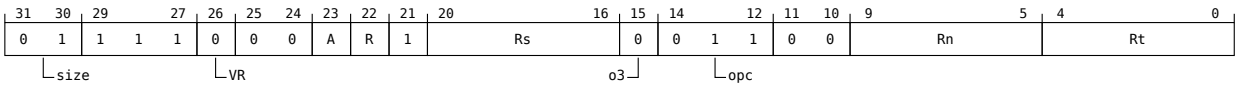
- If the destination register is not WZR, `LDSETAH` and `LDSETALH` load from memory with acquire semantics.
- `LDSETLH` and `LDSETALH` store to memory with release semantics.
- `LDSETH` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STSETH, STSETLH.

Integer
(FEAT_LSE)



Encoding for the No memory ordering variant Applies when `(A == 0 && R == 0)`

```
LDSETH <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire variant Applies when `(A == 1 && R == 0)`

```
LDSETAH <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire-release variant Applies when `(A == 1 && R == 1)`

```
LDSETALH <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Release variant Applies when `(A == 0 && R == 1)`

```
LDSETLH <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
```

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias Conditions

Alias	Is preferred when
STSETH, STSETLH	A == '0' && Rt == '11111'

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_ORR, acquire,
                                                          release, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(16) comparevalue = bits(16) UNKNOWN; // Irrelevant when not executing CAS
constant bits(16) value = X[s, 16];
constant bits(16) data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.139 LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL

Atomic signed maximum on word or doubleword

This instruction atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

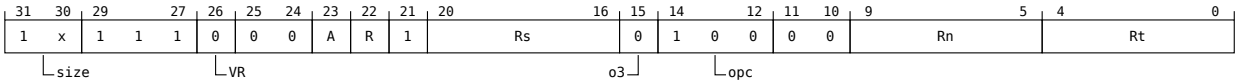
- If the destination register is not one of WZR or XZR, `LDSMAXA` and `LDSMAXAL` load from memory with acquire semantics.
- `LDSMAXL` and `LDSMAXAL` store to memory with release semantics.
- `LDSMAX` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STSMAX, STSMAXL.

Integer
(FEAT_LSE)



Encoding for the 32-bit no memory ordering variant Applies when `(size == 10 && A == 0 && R == 0)`

```
LDSMAX <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire variant Applies when `(size == 10 && A == 1 && R == 0)`

```
LDSMAXA <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire-release variant Applies when `(size == 10 && A == 1 && R == 1)`

```
LDSMAXAL <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit release variant Applies when `(size == 10 && A == 0 && R == 1)`

```
LDSMAXL <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 64-bit no memory ordering variant Applies when `(size == 11 && A == 0 && R == 0)`

```
LDSMAX <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit acquire variant Applies when `(size == 11 && A == 1 && R == 0)`

```
LDSMAXA <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit acquire-release variant Applies when `(size == 11 && A == 1 && R == 1)`

```
LDSMAXAL <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit release variant Applies when `(size == 11 && A == 0 && R == 1)`

```
LDSMAXL <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer datasize = 8 << UInt(size);
constant integer regsize = if datasize == 64 then 64 else 32;

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
```

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

Alias Conditions

Alias	Is preferred when
STSMAX, STSMAXL	A == '0' && Rt == '11111'

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_SMAX, acquire,
                                                          release, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(datasize) comparevalue = bits(datasize) UNKNOWN; // Irrelevant when not executing CAS
constant bits(datasize) value = X[s, datasize];
constant bits(datasize) data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, regsize] = ZeroExtend(data, regsize);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.140 LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB

Atomic signed maximum on byte

This instruction atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

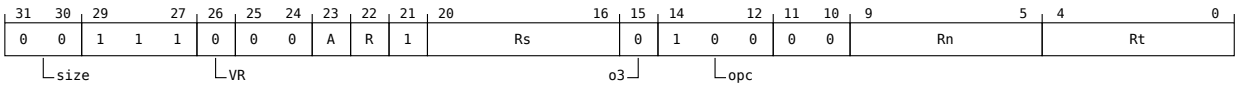
- If the destination register is not WZR, `LDSMAXAB` and `LDSMAXALB` load from memory with acquire semantics.
- `LDSMAXLB` and `LDSMAXALB` store to memory with release semantics.
- `LDSMAXB` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STSMAXB, STSMAXLB.

Integer
(FEAT_LSE)



Encoding for the No memory ordering variant Applies when `(A == 0 && R == 0)`

```
LDSMAXB <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire variant Applies when `(A == 1 && R == 0)`

```
LDSMAXAB <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire-release variant Applies when `(A == 1 && R == 1)`

```
LDSMAXALB <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Release variant Applies when `(A == 0 && R == 1)`

```
LDSMAXLB <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
```

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias Conditions

Alias	Is preferred when
STSMAXB, STSMAXLB	A == '0' && Rt == '11111'

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_SMAX, acquire,
                                                          release, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(8) comparevalue = bits(8) UNKNOWN; // Irrelevant when not executing CAS
constant bits(8) value = X[s, 8];
constant bits(8) data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.141 LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH

Atomic signed maximum on halfword

This instruction atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

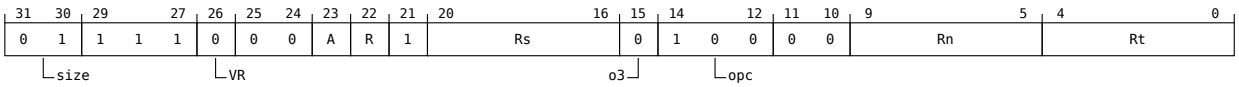
- If the destination register is not WZR, `LDSMAXAH` and `LDSMAXALH` load from memory with acquire semantics.
- `LDSMAXLH` and `LDSMAXALH` store to memory with release semantics.
- `LDSMAXH` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STSMAXH, STSMAXLH.

Integer
(FEAT_LSE)



Encoding for the No memory ordering variant Applies when `(A == 0 && R == 0)`

```
LDSMAXH <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire variant Applies when `(A == 1 && R == 0)`

```
LDSMAXAH <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire-release variant Applies when `(A == 1 && R == 1)`

```
LDSMAXALH <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Release variant Applies when `(A == 0 && R == 1)`

```
LDSMAXLH <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
```

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias Conditions

Alias	Is preferred when
STSMAXH, STSMAXLH	A == '0' && Rt == '11111'

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_SMAX, acquire,
                                                          release, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(16) comparevalue = bits(16) UNKNOWN; // Irrelevant when not executing CAS
constant bits(16) value = X[s, 16];
constant bits(16) data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.142 LDSMIN, LDSMINA, LDSMINAL, LDSMINL

Atomic signed minimum on word or doubleword

This instruction atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

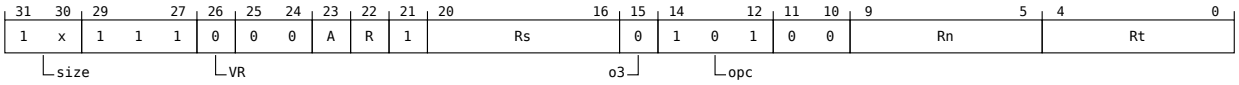
- If the destination register is not one of WZR or XZR, `LDSMINA` and `LDSMINAL` load from memory with acquire semantics.
- `LDSMINL` and `LDSMINAL` store to memory with release semantics.
- `LDSMIN` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STSMIN, STSMINL.

Integer
(FEAT_LSE)



Encoding for the 32-bit no memory ordering variant Applies when `(size == 10 && A == 0 && R == 0)`

```
LDSMIN <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire variant Applies when `(size == 10 && A == 1 && R == 0)`

```
LDSMINA <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire-release variant Applies when `(size == 10 && A == 1 && R == 1)`

```
LDSMINAL <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit release variant Applies when `(size == 10 && A == 0 && R == 1)`

```
LDSMINL <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 64-bit no memory ordering variant Applies when `(size == 11 && A == 0 && R == 0)`

```
LDSMIN <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit acquire variant Applies when `(size == 11 && A == 1 && R == 0)`

```
LDSMINA <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit acquire-release variant Applies when `(size == 11 && A == 1 && R == 1)`

```
LDSMINAL <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit release variant Applies when `(size == 11 && A == 0 && R == 1)`

```
LDSMINL <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer datasize = 8 << UInt(size);
constant integer regsize = if datasize == 64 then 64 else 32;

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
```

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

Alias Conditions

Alias	Is preferred when
STSMIN, STSMINL	A == '0' && Rt == '11111'

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_SMIN, acquire,
                                                          release, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(datasize) comparevalue = bits(datasize) UNKNOWN; // Irrelevant when not executing CAS
constant bits(datasize) value = X[s, datasize];
constant bits(datasize) data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, regsize] = ZeroExtend(data, regsize);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.143 LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB

Atomic signed minimum on byte

This instruction atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

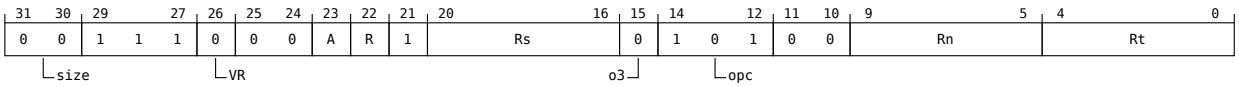
- If the destination register is not WZR, `LDSMINAB` and `LDSMINALB` load from memory with acquire semantics.
- `LDSMINLB` and `LDSMINALB` store to memory with release semantics.
- `LDSMINB` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STSMINB, STSMINLB.

Integer
(FEAT_LSE)



Encoding for the No memory ordering variant Applies when `(A == 0 && R == 0)`

```
LDSMINB <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire variant Applies when `(A == 1 && R == 0)`

```
LDSMINAB <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire-release variant Applies when `(A == 1 && R == 1)`

```
LDSMINALB <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Release variant Applies when `(A == 0 && R == 1)`

```
LDSMINLB <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
```

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias Conditions

Alias	Is preferred when
STSMINB, STSMINLB	A == '0' && Rt == '11111'

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_SMIN, acquire,
                                                          release, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(8) comparevalue = bits(8) UNKNOWN; // Irrelevant when not executing CAS
constant bits(8) value = X[s, 8];
constant bits(8) data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.144 LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH

Atomic signed minimum on halfword

This instruction atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

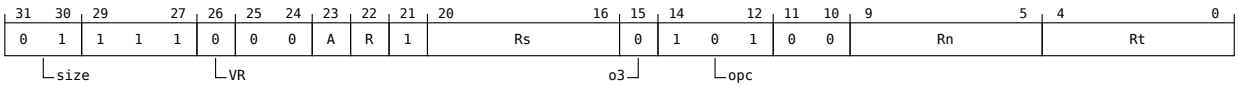
- If the destination register is not WZR, `LDSMINAH` and `LDSMINALH` load from memory with acquire semantics.
- `LDSMINLH` and `LDSMINALH` store to memory with release semantics.
- `LDSMINH` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STSMINH, STSMINLH.

Integer
(FEAT_LSE)



Encoding for the No memory ordering variant Applies when `(A == 0 && R == 0)`

```
LDSMINH <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire variant Applies when `(A == 1 && R == 0)`

```
LDSMINAH <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire-release variant Applies when `(A == 1 && R == 1)`

```
LDSMINALH <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Release variant Applies when `(A == 0 && R == 1)`

```
LDSMINLH <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
```

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias Conditions

Alias	Is preferred when
STSMINH, STSMINLH	A == '0' && Rt == '11111'

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_SMIN, acquire,
                                                          release, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(16) comparevalue = bits(16) UNKNOWN; // Irrelevant when not executing CAS
constant bits(16) value = X[s, 16];
constant bits(16) data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

Operational information

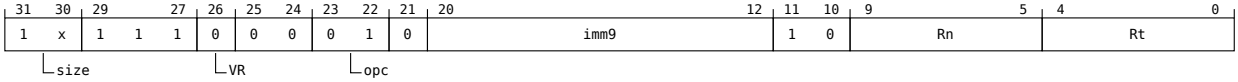
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.145 LDTR

Load register (unprivileged)

This instruction loads a word or doubleword from memory, and writes it to a register. The address that is used for the load is calculated from a base register and an immediate offset.



Encoding for the 32-bit variant Applies when `(size == 10)`

```
LDTR <Wt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 64-bit variant Applies when `(size == 11)`

```
LDTR <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
constant integer scale = UInt(size);
constant bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<**Wt**> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<**simm**> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

<**Xt**> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer datasize = 8 << scale;
constant integer regsize = if datasize == 64 then 64 else 32;
constant boolean nontemporal = FALSE;
```

Operation

```
bits(64) address;

constant boolean privileged = AArch64.IsUnprivAccessPriv();
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

constant bits(datasize) data = Mem[address, datasize DIV 8, accdesc];
X[t, regsize] = ZeroExtend(data, regsize);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational information

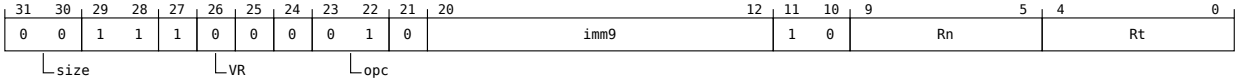
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.146 LDTRB

Load register byte (unprivileged)

This instruction loads a byte from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.



Encoding

```
LDTRB <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
constant bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<**Wt**> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<**simm**> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer datasize = 8;
constant boolean nontemporal = FALSE;
```

Operation

```
bits(64) address;

constant boolean privileged = AArch64.IsUnprivAccessPriv();
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

constant bits(datasize) data = Mem[address, datasize DIV 8, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

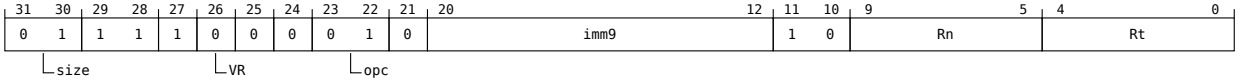
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.147 LDTRH

Load register halfword (unprivileged)

This instruction loads a halfword from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.



Encoding

```
LDTRH <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
constant bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<**Wt**> Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<**simm**> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer datasize = 16;
constant boolean nontemporal = FALSE;
```

Operation

```
bits(64) address;

constant boolean privileged = AArch64.IsUnprivAccessPriv();
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

constant bits(datasize) data = Mem[address, datasize DIV 8, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

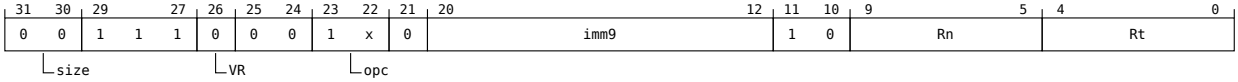
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.148 LDTRSB

Load register signed byte (unprivileged)

This instruction loads a byte from memory, sign-extends it to 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.



Encoding for the 32-bit variant Applies when `(opc == 11)`

```
LDTRSB <Wt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 64-bit variant Applies when `(opc == 10)`

```
LDTRSB <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
constant bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<**Wt**> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<**simm**> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

<**Xt**> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer datasize = 8;
constant integer regsize = 64 >> UInt(opc<0>);
constant boolean nontemporal = FALSE;
```

Operation

```
bits(64) address;

constant boolean privileged = AArch64.IsUnprivAccessPriv();
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

constant bits(datasize) data = Mem[address, datasize DIV 8, accdesc];
X[t, regsize] = SignExtend(data, regsize);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational information

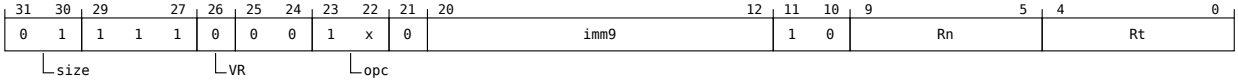
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.149 LDTRSH

Load register signed halfword (unprivileged)

This instruction loads a halfword from memory, sign-extends it to 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.



Encoding for the 32-bit variant Applies when `(opc == 11)`

```
LDTRSH <Wt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 64-bit variant Applies when `(opc == 10)`

```
LDTRSH <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
constant bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<**Wt**> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<**simm**> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

<**Xt**> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer datasize = 16;
constant integer regsize = 64 >> UInt(opc<0>);
constant boolean nontemporal = FALSE;
```

Operation

```
bits(64) address;

constant boolean privileged = AArch64.IsUnprivAccessPriv();
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

constant bits(datasize) data = Mem[address, datasize DIV 8, accdesc];
X[t, regsize] = SignExtend(data, regsize);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational information

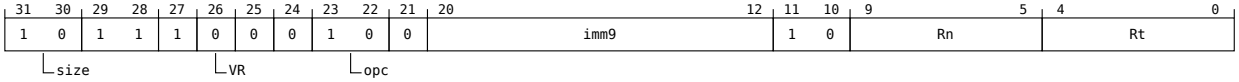
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.150 LDTRSW

Load register signed word (unprivileged)

This instruction loads a word from memory, sign-extends it to 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.



Encoding

```
LDTRSW <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
constant bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer datasize = 32;
constant boolean nontemporal = FALSE;
```

Operation

```
bits(64) address;

constant boolean privileged = AArch64.IsUnprivAccessPriv();
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

constant bits(datasize) data = Mem[address, datasize DIV 8, accdesc];
X[t, 64] = SignExtend(data, 64);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.151 LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL

Atomic unsigned maximum on word or doubleword

This instruction atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

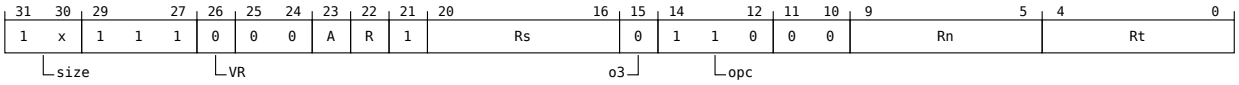
- If the destination register is not one of WZR or XZR, `LDUMAXA` and `LDUMAXAL` load from memory with acquire semantics.
- `LDUMAXL` and `LDUMAXAL` store to memory with release semantics.
- `LDUMAX` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STUMAX, STUMAXL.

Integer
(FEAT_LSE)



Encoding for the 32-bit no memory ordering variant Applies when `(size == 10 && A == 0 && R == 0)`

```
LDUMAX <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire variant Applies when `(size == 10 && A == 1 && R == 0)`

```
LDUMAXA <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire-release variant Applies when `(size == 10 && A == 1 && R == 1)`

```
LDUMAXAL <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit release variant Applies when `(size == 10 && A == 0 && R == 1)`

```
LDUMAXL <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 64-bit no memory ordering variant Applies when `(size == 11 && A == 0 && R == 0)`

```
LDUMAX <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit acquire variant Applies when `(size == 11 && A == 1 && R == 0)`

```
LDUMAXA <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit acquire-release variant Applies when `(size == 11 && A == 1 && R == 1)`

```
LDUMAXAL <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit release variant Applies when `(size == 11 && A == 0 && R == 1)`

```
LDUMAXL <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer datasize = 8 << UInt(size);
constant integer regsize = if datasize == 64 then 64 else 32;

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
```

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

Alias Conditions

Alias	Is preferred when
STUMAX, STUMAXL	A == '0' && Rt == '11111'

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_UMAX, acquire,
                                                          release, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(datasize) comparevalue = bits(datasize) UNKNOWN; // Irrelevant when not executing CAS
constant bits(datasize) value = X[s, datasize];
constant bits(datasize) data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, regsize] = ZeroExtend(data, regsize);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.152 LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB

Atomic unsigned maximum on byte

This instruction atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

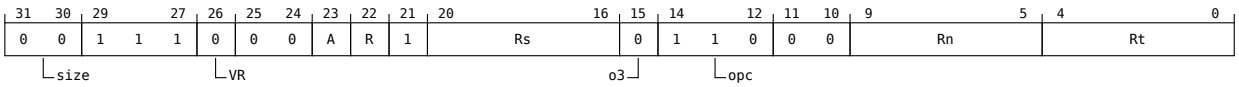
- If the destination register is not WZR, `LDUMAXAB` and `LDUMAXALB` load from memory with acquire semantics.
- `LDUMAXLB` and `LDUMAXALB` store to memory with release semantics.
- `LDUMAXB` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STUMAXB, STUMAXLB.

Integer
(FEAT_LSE)



Encoding for the No memory ordering variant Applies when `(A == 0 && R == 0)`

```
LDUMAXB <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire variant Applies when `(A == 1 && R == 0)`

```
LDUMAXAB <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire-release variant Applies when `(A == 1 && R == 1)`

```
LDUMAXALB <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Release variant Applies when `(A == 0 && R == 1)`

```
LDUMAXLB <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
```

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias Conditions

Alias	Is preferred when
STUMAXB, STUMAXLB	A == '0' && Rt == '11111'

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_UMAX, acquire,
                                                          release, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(8) comparevalue = bits(8) UNKNOWN; // Irrelevant when not executing CAS
constant bits(8) value = X[s, 8];
constant bits(8) data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.153 LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH

Atomic unsigned maximum on halfword

This instruction atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

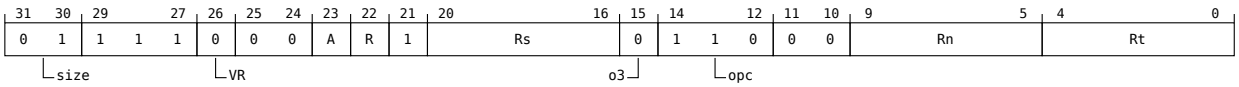
- If the destination register is not WZR, `LDUMAXAH` and `LDUMAXALH` load from memory with acquire semantics.
- `LDUMAXLH` and `LDUMAXALH` store to memory with release semantics.
- `LDUMAXH` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STUMAXH, STUMAXLH.

Integer
(FEAT_LSE)



Encoding for the No memory ordering variant Applies when `(A == 0 && R == 0)`

```
LDUMAXH <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire variant Applies when `(A == 1 && R == 0)`

```
LDUMAXAH <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire-release variant Applies when `(A == 1 && R == 1)`

```
LDUMAXALH <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Release variant Applies when `(A == 0 && R == 1)`

```
LDUMAXLH <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
```

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias Conditions

Alias	Is preferred when
STUMAXH, STUMAXLH	A == '0' && Rt == '11111'

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_UMAX, acquire,
                                                          release, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(16) comparevalue = bits(16) UNKNOWN; // Irrelevant when not executing CAS
constant bits(16) value = X[s, 16];
constant bits(16) data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.154 LDUMIN, LDUMINA, LDUMINAL, LDUMINL

Atomic unsigned minimum on word or doubleword

This instruction atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

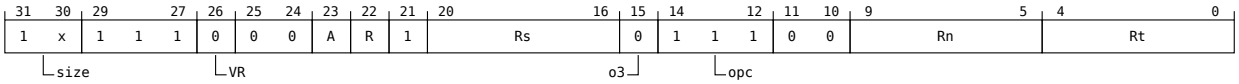
- If the destination register is not one of WZR or XZR, `LDUMINA` and `LDUMINAL` load from memory with acquire semantics.
- `LDUMINL` and `LDUMINAL` store to memory with release semantics.
- `LDUMIN` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STUMIN, STUMINL.

Integer
(FEAT_LSE)



Encoding for the 32-bit no memory ordering variant Applies when `(size == 10 && A == 0 && R == 0)`

```
LDUMIN <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire variant Applies when `(size == 10 && A == 1 && R == 0)`

```
LDUMINA <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit acquire-release variant Applies when `(size == 10 && A == 1 && R == 1)`

```
LDUMINAL <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit release variant Applies when `(size == 10 && A == 0 && R == 1)`

```
LDUMINL <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 64-bit no memory ordering variant Applies when `(size == 11 && A == 0 && R == 0)`

```
LDUMIN <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit acquire variant Applies when `(size == 11 && A == 1 && R == 0)`

```
LDUMINA <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit acquire-release variant Applies when `(size == 11 && A == 1 && R == 1)`

```
LDUMINAL <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit release variant Applies when `(size == 11 && A == 0 && R == 1)`

```
LDUMINL <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer datasize = 8 << UInt(size);
constant integer regsize = if datasize == 64 then 64 else 32;

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
```

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

Alias Conditions

Alias	Is preferred when
STUMIN, STUMINL	A == '0' && Rt == '11111'

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_UMIN, acquire,
                                                          release, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(datasize) comparevalue = bits(datasize) UNKNOWN; // Irrelevant when not executing CAS
constant bits(datasize) value = X[s, datasize];
constant bits(datasize) data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, regsize] = ZeroExtend(data, regsize);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.155 LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB

Atomic unsigned minimum on byte

This instruction atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

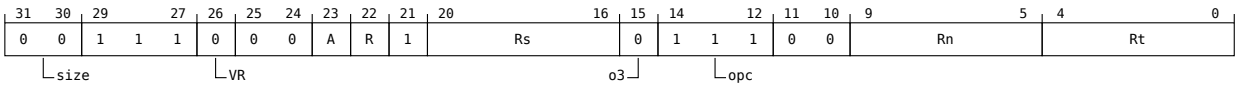
- If the destination register is not WZR, `LDUMINAB` and `LDUMINALB` load from memory with acquire semantics.
- `LDUMINLB` and `LDUMINALB` store to memory with release semantics.
- `LDUMINB` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STUMINB, STUMINLB.

Integer
(FEAT_LSE)



Encoding for the No memory ordering variant Applies when `(A == 0 && R == 0)`

```
LDUMINB <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire variant Applies when `(A == 1 && R == 0)`

```
LDUMINAB <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire-release variant Applies when `(A == 1 && R == 1)`

```
LDUMINALB <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Release variant Applies when `(A == 0 && R == 1)`

```
LDUMINLB <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
```

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias Conditions

Alias	Is preferred when
STUMINB, STUMINLB	A == '0' && Rt == '11111'

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_UMIN, acquire,
                                                          release, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(8) comparevalue = bits(8) UNKNOWN; // Irrelevant when not executing CAS
constant bits(8) value = X[s, 8];
constant bits(8) data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.156 LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH

Atomic unsigned minimum on halfword

This instruction atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

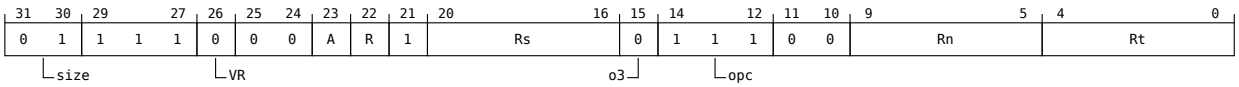
- If the destination register is not WZR, `LDUMINAH` and `LDUMINALH` load from memory with acquire semantics.
- `LDUMINLH` and `LDUMINALH` store to memory with release semantics.
- `LDUMINH` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This instruction is used by the alias STUMINH, STUMINLH.

Integer
(FEAT_LSE)



Encoding for the No memory ordering variant Applies when `(A == 0 && R == 0)`

```
LDUMINH <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire variant Applies when `(A == 1 && R == 0)`

```
LDUMINAH <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Acquire-release variant Applies when `(A == 1 && R == 1)`

```
LDUMINALH <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the Release variant Applies when `(A == 0 && R == 1)`

```
LDUMINLH <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
```

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias Conditions

Alias	Is preferred when
STUMINH, STUMINLH	A == '0' && Rt == '11111'

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_UMIN, acquire,
                                                          release, privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

constant bits(16) comparevalue = bits(16) UNKNOWN; // Irrelevant when not executing CAS
constant bits(16) value = X[s, 16];
constant bits(16) data = MemAtomic(address, comparevalue, value, accdesc);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

Operational information

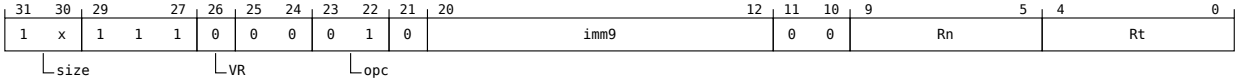
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.157 LDUR

Load register (unscaled)

This instruction calculates an address from a base register and an immediate offset, loads a 32-bit word or 64-bit doubleword from memory, zero-extends it, and writes it to a register. For information about addressing modes, see Load/Store addressing modes.



Encoding for the 32-bit variant Applies when `(size == 10)`

```
LDUR <Wt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 64-bit variant Applies when `(size == 11)`

```
LDUR <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
constant integer scale = UInt(size);
constant bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<**Wt**> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<**simm**> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

<**Xt**> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

Shared Decode

```
constant integer n = UInt(Rn);
constant integer t = UInt(Rt);
constant integer datasize = 8 << scale;
constant integer regsize = if datasize == 64 then 64 else 32;
constant boolean nontemporal = FALSE;
```

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

constant bits(datasize) data = Mem[address, datasize DIV 8, accdesc];
X[t, regsize] = ZeroExtend(data, regsize);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational information

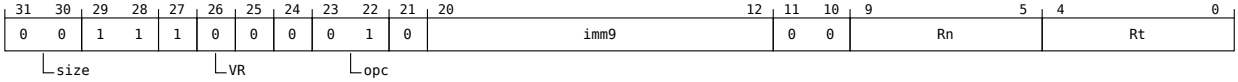
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.158 LDURB

Load register byte (unscaled)

This instruction calculates an address from a base register and an immediate offset, loads a byte from memory, zero-extends it, and writes it to a register. For information about addressing modes, see Load/Store addressing modes.



Encoding

```
LDURB <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
constant bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<**Wt**> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<**simm**> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
constant integer n = UInt(Rn);
constant integer t = UInt(Rt);
constant integer datasize = 8;
constant boolean nontemporal = FALSE;
```

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

constant bits(datasize) data = Mem[address, datasize DIV 8, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

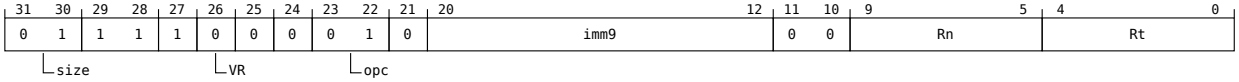
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.159 LDURH

Load register halfword (unscaled)

This instruction calculates an address from a base register and an immediate offset, loads a halfword from memory, zero-extends it, and writes it to a register. For information about addressing modes, see Load/Store addressing modes.



Encoding

```
LDURH <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
constant bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<**Wt**> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<**simm**> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
constant integer n = UInt(Rn);
constant integer t = UInt(Rt);
constant integer datasize = 16;
constant boolean nontemporal = FALSE;
```

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

constant bits(datasize) data = Mem[address, datasize DIV 8, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

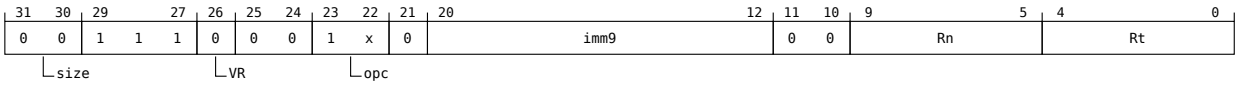
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.160 LDURSB

Load register signed byte (unscaled)

This instruction calculates an address from a base register and an immediate offset, loads a signed byte from memory, sign-extends it, and writes it to a register. For information about addressing modes, see Load/Store addressing modes.



Encoding for the 32-bit variant Applies when `(opc == 11)`

```
LDURSB <Wt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 64-bit variant Applies when `(opc == 10)`

```
LDURSB <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
constant bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

Shared Decode

```
constant integer n = UInt(Rn);
constant integer t = UInt(Rt);
constant integer datasize = 8;
constant integer regsize = 64 >> UInt(opc<0>);
constant boolean nontemporal = FALSE;
```

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

constant bits(datasize) data = Mem[address, datasize DIV 8, accdesc];
X[t, regsize] = SignExtend(data, regsize);
```

Operational information

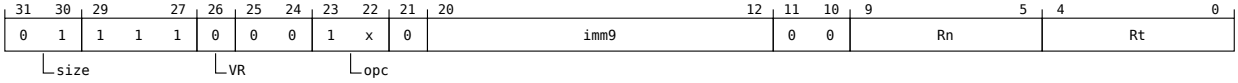
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.161 LDURSH

Load register signed halfword (unscaled)

This instruction calculates an address from a base register and an immediate offset, loads a signed halfword from memory, sign-extends it, and writes it to a register. For information about addressing modes, see Load/Store addressing modes.



Encoding for the 32-bit variant Applies when `(opc == 11)`

```
LDURSH <Wt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 64-bit variant Applies when `(opc == 10)`

```
LDURSH <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
constant bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<**Wt**> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<**simm**> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

<**Xt**> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

Shared Decode

```
constant integer n = UInt(Rn);
constant integer t = UInt(Rt);
constant integer datasize = 16;
constant integer regsize = 64 >> UInt(opc<0>);
constant boolean nontemporal = FALSE;
```

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

constant bits(datasize) data = Mem[address, datasize DIV 8, accdesc];
X[t, regsize] = SignExtend(data, regsize);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational information

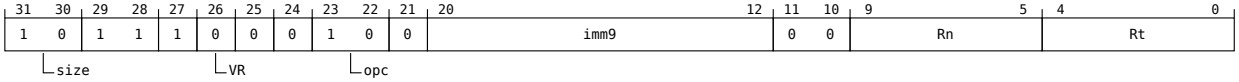
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.162 LDURSW

Load register signed word (unscaled)

This instruction calculates an address from a base register and an immediate offset, loads a signed word from memory, sign-extends it, and writes it to a register. For information about addressing modes, see Load/Store addressing modes.



Encoding

```
LDURSW <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
constant bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
constant integer n = UInt(Rn);
constant integer t = UInt(Rt);
constant integer datasize = 32;
constant boolean nontemporal = FALSE;
```

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

constant bits(datasize) data = Mem[address, datasize DIV 8, accdesc];
X[t, 64] = SignExtend(data, 64);
```

Operational information

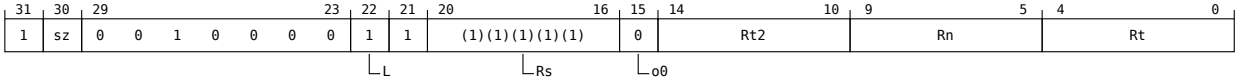
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.163 LDXP

Load exclusive pair of registers

This instruction derives an address from a base register value, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. For information on single-copy atomicity and alignment requirements, see Requirements for single-copy atomicity and Alignment of data accesses. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores. For information about addressing modes, see Load/Store addressing modes.



Encoding for the 32-bit variant Applies when `(sz == 0)`

```
LDXP <Wt1>, <Wt2>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit variant Applies when `(sz == 1)`

```
LDXP <Xt1>, <Xt2>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
constant integer t = UInt(Rt);
constant integer t2 = UInt(Rt2);
constant integer n = UInt(Rn);

constant integer elsize = 32 << UInt(sz);
constant integer datasize = elsize * 2;
constant boolean acqrel = FALSE;

boolean rt_unknown = FALSE;
if t == t2 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDXP.

Assembler Symbols

<**Wt1**> Is the 32-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<**Wt2**> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<**Xt1**> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.

<**Xt2**> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the “Rt2” field.

Operation

```
bits(64) address;
bits(datasize) data;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer dbytes = datasize DIV 8;
constant boolean privileged = PSTATE.EL != EL0;
constant boolean ispair = elsize == 64;   // When elsize is 32, the access is single-copy atomic
constant AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_LOAD, acqrel, privileged, ispair, t,
                                                        t2);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

AArch64.SetExclusiveMonitors(address, dbytes, accdesc);

data = Mem[address, dbytes, accdesc];
if rt_unknown then
    // ConstrainedUNPREDICTABLE case
    X[t, datasize]  = bits(datasize) UNKNOWN;        // In this case t = t2
elsif BigEndian(accdesc.acctype) then
    X[t, elsize]  = data<elsize +: elsize>;
    X[t2, elsize] = data<0 +: elsize>;
else
    X[t, elsize]  = data<0 +: elsize>;
    X[t2, elsize] = data<elsize +: elsize>;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

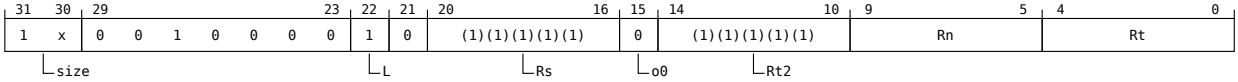
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.164 LDXR

Load exclusive register

This instruction derives an address from a base register value, loads a 32-bit word or a 64-bit doubleword from memory, and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores.

For information about addressing modes, see Load/Store addressing modes.



Encoding for the 32-bit variant Applies when `(size == 10)`

```
LDXR <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit variant Applies when `(size == 11)`

```
LDXR <Xt>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer elsize = 8 << UInt(size);
constant integer regsize = if elsize == 64 then 64 else 32;
constant boolean acqrel = FALSE;
```

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

Operation

```
bits(64) address;
bits(elsize) data;

constant integer dbytes = elsize DIV 8;
constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_LOAD, acqrel, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

AArch64.SetExclusiveMonitors(address, dbytes, accdesc);

data = Mem[address, dbytes, accdesc];
X[t, regsize] = ZeroExtend(data, regsize);
```

Operational information

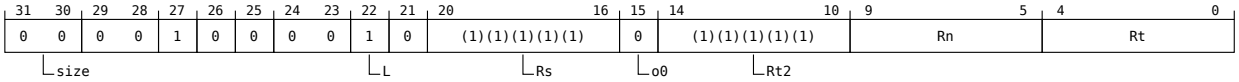
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.165 LDXRB

Load exclusive register byte

This instruction derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores. For information about addressing modes, see Load/Store addressing modes.



Encoding

```
LDXRB <Wt>, [<Xn|SP>{, #0}]
```

Decode for this encoding

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acqrel = FALSE;
```

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) data;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_LOAD, acqrel, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

AArch64.SetExclusiveMonitors(address, 1, accdesc);

data = Mem[address, 1, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

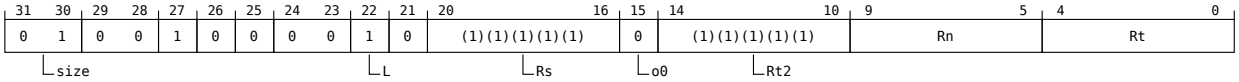
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.166 LDXRH

Load exclusive register halfword

This instruction derives an address from a base register value, loads a halfword from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores. For information about addressing modes, see Load/Store addressing modes.



Encoding

```
LDXRH <Wt>, [<Xn|SP>{, #0}]
```

Decode for this encoding

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acqrel = FALSE;
```

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) data;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_LOAD, acqrel, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

AArch64.SetExclusiveMonitors(address, 2, accdesc);

data = Mem[address, 2, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

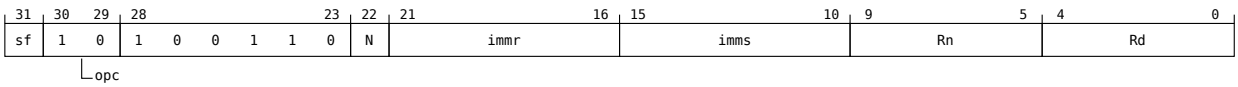
C6.2.167 LSL (immediate)

Logical shift left (immediate)

This instruction shifts a register value left by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

This is an alias of UBFM. This means:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding for the 32-bit variant Applies when `(sf == 0 && N == 0 && imms != 011111)`

```
LSL <Wd>, <Wn>, #<shift>
```

is equivalent to

```
UBFM <Wd>, <Wn>, #(-<shift> MOD 32), #(31-<shift>)
```

and is the preferred disassembly when `UInt(imms) + 1 == UInt(immr)`.

Encoding for the 64-bit variant Applies when `(sf == 1 && N == 1 && imms != 111111)`

```
LSL <Xd>, <Xn>, #<shift>
```

is equivalent to

```
UBFM <Xd>, <Xn>, #(-<shift> MOD 64), #(63-<shift>)
```

and is the preferred disassembly when `UInt(imms) + 1 == UInt(immr)`.

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wn**> Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<**shift**> For the “32-bit” variant: is the shift amount, in the range 0 to 31.

For the “64-bit” variant: is the shift amount, in the range 0 to 63.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Xn**> Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

The description of UBFM gives the operational pseudocode for this instruction.

Operational Information

The description of UBFM gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

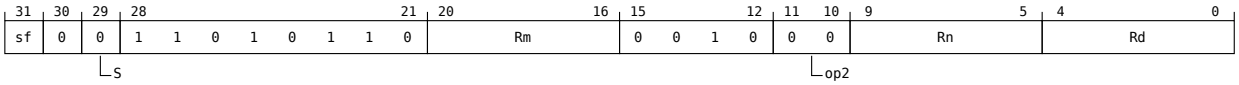
C6.2.168 LSL (register)

Logical shift left (register)

This instruction shifts a register value left by a variable number of bits, shifting in zeros, and writes the result to the destination register. The value of the second source register modulo the register size in bits gives the number of bits by which the first source register is left-shifted.

This is an alias of LSLV. This means:

- The encodings in this description are named to match the encodings of LSLV.
- The description of LSLV gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
LSL <Wd>, <Wn>, <Wm>
```

is equivalent to

```
LSLV <Wd>, <Wn>, <Wm>
```

and is always the preferred disassembly.

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
LSL <Xd>, <Xn>, <Xm>
```

is equivalent to

```
LSLV <Xd>, <Xn>, <Xm>
```

and is always the preferred disassembly.

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wn**> Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Wm**> Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the “Rm” field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Xn**> Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Xm**> Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the “Rm” field.

Operation

The description of LSLV gives the operational pseudocode for this instruction.

Operational Information

The description of LSLV gives the operational information for this instruction.

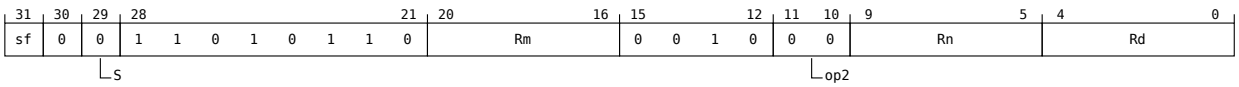
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.169 LSLV

Logical shift left variable

This instruction shifts a register value left by a variable number of bits, shifting in zeros, and writes the result to the destination register. The value of the second source register modulo the register size in bits gives the number of bits by which the first source register is left-shifted.

This instruction is used by the alias LSL (register).



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
LSLV <Wd>, <Wn>, <Wm>
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
LSLV <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
constant ShiftType shift_type = DecodeShift(op2);
```

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wn**> Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Wm**> Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the “Rm” field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Xn**> Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Xm**> Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the “Rm” field.

Operation

```
constant bits(datasize) operand2 = X[m, datasize];

X[d, datasize] = ShiftReg(n, shift_type, UInt(operand2) MOD datasize, datasize);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

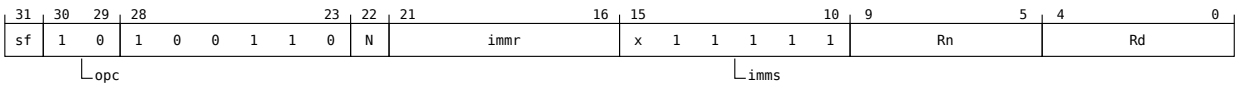
C6.2.170 LSR (immediate)

Logical shift right (immediate)

This instruction shifts a register value right by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

This is an alias of UBFM. This means:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding for the 32-bit variant Applies when `(sf == 0 && N == 0 && imms == 011111)`

```
LSR <Wd>, <Wn>, #<shift>
```

is equivalent to

```
UBFM <Wd>, <Wn>, #<shift>, #31
```

and is always the preferred disassembly.

Encoding for the 64-bit variant Applies when `(sf == 1 && N == 1 && imms == 111111)`

```
LSR <Xd>, <Xn>, #<shift>
```

is equivalent to

```
UBFM <Xd>, <Xn>, #<shift>, #63
```

and is always the preferred disassembly.

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wn**> Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<**shift**> For the “32-bit” variant: is the shift amount, in the range 0 to 31, encoded in the “immr” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, encoded in the “immr” field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Xn**> Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

The description of UBFM gives the operational pseudocode for this instruction.

Operational Information

The description of UBFM gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

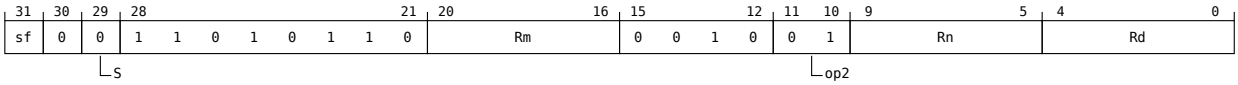
C6.2.171 LSR (register)

Logical shift right (register)

This instruction shifts a register value right by a variable number of bits, shifting in zeros, and writes the result to the destination register. The value of the second source register modulo the register size in bits gives the number of bits by which the first source register is right-shifted.

This is an alias of LSRV. This means:

- The encodings in this description are named to match the encodings of LSRV.
- The description of LSRV gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
LSR <Wd>, <Wn>, <Wm>
```

is equivalent to

```
LSRV <Wd>, <Wn>, <Wm>
```

and is always the preferred disassembly.

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
LSR <Xd>, <Xn>, <Xm>
```

is equivalent to

```
LSRV <Xd>, <Xn>, <Xm>
```

and is always the preferred disassembly.

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wn**> Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Wm**> Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the “Rm” field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Xn**> Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Xm**> Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the “Rm” field.

Operation

The description of LSRV gives the operational pseudocode for this instruction.

Operational Information

The description of LSRV gives the operational information for this instruction.

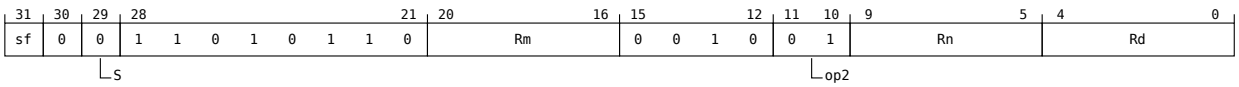
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.172 LSRV

Logical shift right variable

This instruction shifts a register value right by a variable number of bits, shifting in zeros, and writes the result to the destination register. The value of the second source register modulo the register size in bits gives the number of bits by which the first source register is right-shifted.

This instruction is used by the alias LSR (register).



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
LSRV <Wd>, <Wn>, <Wm>
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
LSRV <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
constant ShiftType shift_type = DecodeShift(op2);
```

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wn**> Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Wm**> Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the “Rm” field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Xn**> Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Xm**> Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the “Rm” field.

Operation

```
constant bits(datasize) operand2 = X[m, datasize];

X[d, datasize] = ShiftReg(n, shift_type, UInt(operand2) MOD datasize, datasize);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

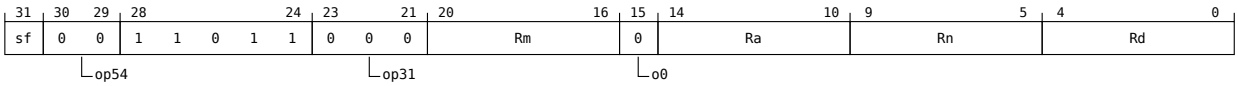
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.173 MADD

Multiply-add

This instruction multiplies two register values, adds a third register value, and writes the result to the destination register.

This instruction is used by the alias MUL.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
MADD <Wd>, <Wn>, <Wm>, <Wa>
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
MADD <Xd>, <Xn>, <Xm>, <Xa>
```

Decode for all variants of this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer a = UInt(Ra);
constant integer datasize = 32 << UInt(sf);
```

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wn**> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<**Wm**> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

<**Wa**> Is the 32-bit name of the third general-purpose source register holding the addend, encoded in the “Ra” field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Xn**> Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<**Xm**> Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

<**Xa**> Is the 64-bit name of the third general-purpose source register holding the addend, encoded in the “Ra” field.

Alias Conditions

Alias	Is preferred when
MUL	`Ra == '11111'`

Operation

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant bits(datasize) operand1 = X[n, datasize];
constant bits(datasize) operand2 = X[m, datasize];
constant bits(datasize) operand3 = X[a, datasize];

constant integer result = UInt(operand3) + (UInt(operand1) * UInt(operand2));

X[d, datasize] = result<datasize-1:0>;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

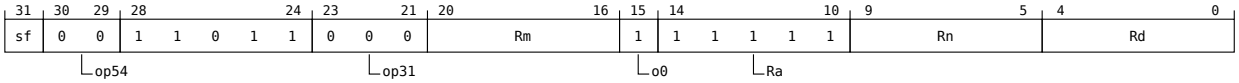
C6.2.174 MNEG

Multiply-negate

This instruction multiplies two register values, negates the product, and writes the result to the destination register.

This is an alias of MSUB. This means:

- The encodings in this description are named to match the encodings of MSUB.
- The description of MSUB gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
MNEG <Wd>, <Wn>, <Wm>
```

is equivalent to

```
MSUB <Wd>, <Wn>, <Wm>, WZR
```

and is always the preferred disassembly.

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
MNEG <Xd>, <Xn>, <Xm>
```

is equivalent to

```
MSUB <Xd>, <Xn>, <Xm>, XZR
```

and is always the preferred disassembly.

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wn**> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<**Wm**> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Xn**> Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<**Xm**> Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

Operation

The description of MSUB gives the operational pseudocode for this instruction.

Operational Information

The description of MSUB gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

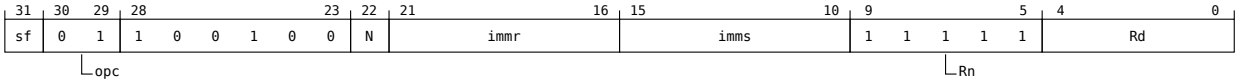
C6.2.175 MOV (bitmask immediate)

Move bitmask immediate value

This instruction writes a bitmask immediate value to a register.

This is an alias of ORR (immediate). This means:

- The encodings in this description are named to match the encodings of ORR (immediate).
- The description of ORR (immediate) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding for the 32-bit variant Applies when `(sf == 0 && N == 0)`

```
MOV <Wd|WSP>, #<imm>
```

is equivalent to

```
ORR <Wd|WSP>, WZR, #<imm>
```

and is the preferred disassembly when `!MoveWidePreferred(sf, N, imms, immr)`.

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
MOV <Xd|SP>, #<imm>
```

is equivalent to

```
ORR <Xd|SP>, XZR, #<imm>
```

and is the preferred disassembly when `!MoveWidePreferred(sf, N, imms, immr)`.

Assembler Symbols

<**Wd**|**WSP**> Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

<**imm**> For the "32-bit" variant: is the bitmask immediate, encoded in "imms:immr", but excluding values which could be encoded by MOVZ or MOVN.

For the "64-bit" variant: is the bitmask immediate, encoded in "N:imms:immr", but excluding values which could be encoded by MOVZ or MOVN.

<**Xd**|**SP**> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

Operation

The description of ORR (immediate) gives the operational pseudocode for this instruction.

Operational Information

The description of ORR (immediate) gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.176 MOV (inverted wide immediate)

Move inverted wide immediate value

This instruction moves an inverted 16-bit immediate value to a register.

This is an alias of MOVN. This means:

- The encodings in this description are named to match the encodings of MOVN.
- The description of MOVN gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Encoding for the 32-bit variant Applies when `(sf == 0 && hw == 0x)`

```
MOV <Wd>, #<imm>
```

is equivalent to

```
MOVN <Wd>, #<imm16>, LSL #<shift>
```

and is the preferred disassembly when `!(IsZero(imm16) && hw != '00') && !IsOnes(imm16).`

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
MOV <Xd>, #<imm>
```

is equivalent to

```
MOVN <Xd>, #<imm16>, LSL #<shift>
```

and is the preferred disassembly when `!(IsZero(imm16) && hw != '00').`

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**imm**> For the "32-bit" variant: is a 32-bit immediate, the bitwise inverse of which can be encoded in "imm16:hw", but excluding 0xFFFF0000 and 0x0000FFFF

For the "64-bit" variant: is a 64-bit immediate, the bitwise inverse of which can be encoded in "imm16:hw".

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

Operation

The description of MOVN gives the operational pseudocode for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

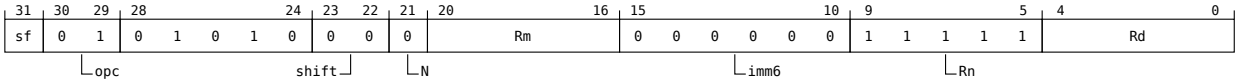
C6.2.177 MOV (register)

Move register value

This instruction copies the value in a source register to the destination register.

This is an alias of ORR (shifted register). This means:

- The encodings in this description are named to match the encodings of ORR (shifted register).
- The description of ORR (shifted register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
MOV <Wd>, <Wm>
```

is equivalent to

```
ORR <Wd>, WZR, <Wm>
```

and is always the preferred disassembly.

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
MOV <Xd>, <Xm>
```

is equivalent to

```
ORR <Xd>, XZR, <Xm>
```

and is always the preferred disassembly.

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wm**> Is the 32-bit name of the general-purpose source register, encoded in the “Rm” field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Xm**> Is the 64-bit name of the general-purpose source register, encoded in the “Rm” field.

Operation

The description of ORR (shifted register) gives the operational pseudocode for this instruction.

Operational Information

The description of ORR (shifted register) gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

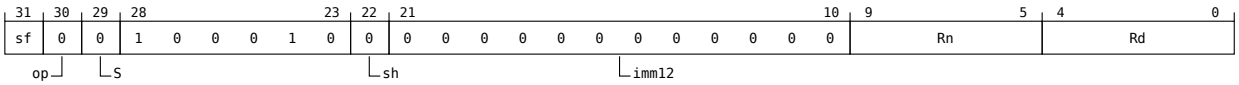
C6.2.178 MOV (to/from SP)

Move register value to or from SP

This instruction copies the value of a register to or from the stack pointer.

This is an alias of ADD (immediate). This means:

- The encodings in this description are named to match the encodings of ADD (immediate).
- The description of ADD (immediate) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
MOV <Wd|WSP>, <Wn|WSP>
```

is equivalent to

```
ADD <Wd|WSP>, <Wn|WSP>, #0
```

and is the preferred disassembly when `Rd == '11111' || Rn == '11111'`.

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
MOV <Xd|SP>, <Xn|SP>
```

is equivalent to

```
ADD <Xd|SP>, <Xn|SP>, #0
```

and is the preferred disassembly when `Rd == '11111' || Rn == '11111'`.

Assembler Symbols

<**Wd**|**WSP**> Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.

<**Wn**|**WSP**> Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the “Rn” field.

<**Xd**|**SP**> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.

<**Xn**|**SP**> Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the “Rn” field.

Operation

The description of ADD (immediate) gives the operational pseudocode for this instruction.

Operational Information

The description of ADD (immediate) gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.179 MOV (wide immediate)

Move wide immediate value

This instruction moves a 16-bit immediate value to a register.

This is an alias of MOVZ. This means:

- The encodings in this description are named to match the encodings of MOVZ.
- The description of MOVZ gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Encoding for the 32-bit variant Applies when `(sf == 0 && hw == 0x)`

```
MOV <Wd>, #<imm>
```

is equivalent to

```
MOVZ <Wd>, #<imm16>, LSL #<shift>
```

and is the preferred disassembly when `!(IsZero(imm16) && hw != '00')`.

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
MOV <Xd>, #<imm>
```

is equivalent to

```
MOVZ <Xd>, #<imm16>, LSL #<shift>
```

and is the preferred disassembly when `!(IsZero(imm16) && hw != '00')`.

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**imm**> For the "32-bit" variant: is a 32-bit immediate which can be encoded in "imm16:hw".

For the "64-bit" variant: is a 64-bit immediate which can be encoded in "imm16:hw".

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

Operation

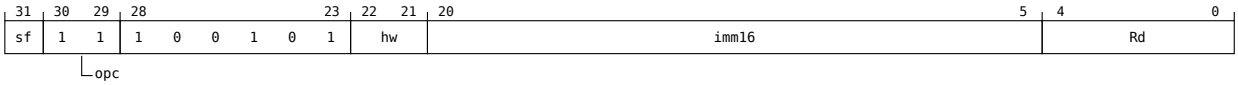
The description of MOVZ gives the operational pseudocode for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.180 MOVK

Move wide with keep

This instruction moves an optionally-shifted 16-bit immediate value into a register, keeping other bits unchanged.



Encoding for the 32-bit variant Applies when `(sf == 0 && hw == 0x)`

```
MOVK <Wd>, #<imm>{, LSL #<shift>}
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
MOVK <Xd>, #<imm>{, LSL #<shift>}
```

Decode for all variants of this encoding

```
if sf == '0' && hw<1> == '1' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer datasize = 32 << UInt(sf);
constant bits(16) imm = imm16;
constant integer pos = UInt(hw) << 4;
```

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**imm**> Is the 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

<**shift**> For the "32-bit" variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16.

For the "64-bit" variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

Operation

```
bits(datasize) result = X[d, datasize];
result<pos+15:pos> = imm;
X[d, datasize] = result;
```

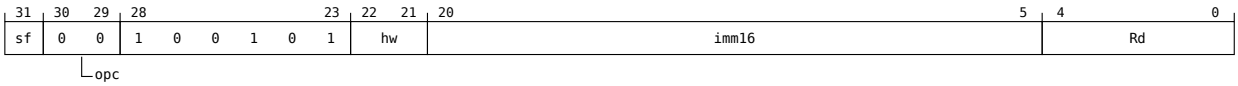

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.181 MOVN

Move wide with NOT

This instruction moves the inverse of an optionally-shifted 16-bit immediate value to a register.

This instruction is used by the alias MOV (inverted wide immediate).



Encoding for the 32-bit variant Applies when `(sf == 0 && hw == 0x)`

```
MOVN <Wd>, #<imm>{, LSL #<shift>}
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
MOVN <Xd>, #<imm>{, LSL #<shift>}
```

Decode for all variants of this encoding

```
if sf == '0' && hw<1> == '1' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer datasize = 32 << UInt(sf);
constant bits(16) imm = imm16;
constant integer pos = UInt(hw) << 4;
```

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**imm**> Is the 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

<**shift**> For the "32-bit" variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16.

For the "64-bit" variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

Alias Conditions

Alias	Of variant	Is preferred when
MOV (inverted wide immediate)	32-bit	`!(IsZero(imm16) && hw != '00') && !IsOnes(imm16)`
MOV (inverted wide immediate)	64-bit	`!(IsZero(imm16) && hw != '00')`

Operation

```
bits(datasize) result = Zeros(datasize);
result<pos+15:pos> = imm;
X[d, datasize] = NOT(result);
```

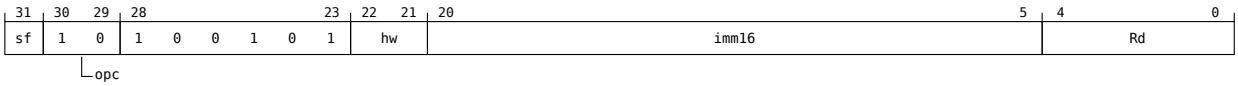

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.182 MOVZ

Move wide with zero

This instruction moves an optionally-shifted 16-bit immediate value to a register.

This instruction is used by the alias MOV (wide immediate).



Encoding for the 32-bit variant Applies when `(sf == 0 && hw == 0x)`

```
MOVZ <Wd>, #<imm>{, LSL #<shift>}
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
MOVZ <Xd>, #<imm>{, LSL #<shift>}
```

Decode for all variants of this encoding

```
if sf == '0' && hw<1> == '1' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer datasize = 32 << UInt(sf);
constant bits(16) imm = imm16;
constant integer pos = UInt(hw) << 4;
```

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**imm**> Is the 16-bit unsigned immediate, in the range 0 to 65535, encoded in the “imm16” field.

<**shift**> For the “32-bit” variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the “hw” field as <shift>/16.

For the “64-bit” variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the “hw” field as <shift>/16.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

Alias Conditions

Alias	Is preferred when
MOV (wide immediate)	!(IsZero(imm16) && hw != '00')

Operation

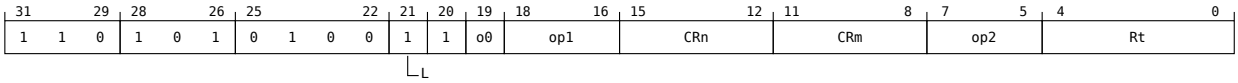
```
bits(datasize) result = Zeros(datasize);
result<pos+15:pos> = imm;
X[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.183 MRS

Move System register to general-purpose register

This instruction allows the PE to read an AArch64 System register into a general-purpose register.



Encoding

```
MRS <Xt>, (<systemreg>|S<op0>_<op1>_<Cn>_<Cm>_<op2>)
```

Decode for this encoding

```
constant integer t       = UInt(Rt);
constant bits(1) sys_L   = L;
constant bits(2) sys_op0 = '1' : o0;
constant bits(3) sys_op1 = op1;
constant bits(3) sys_op2 = op2;
constant bits(4) sys_crn = CRn;
constant bits(4) sys_crm = CRm;
```

Assembler Symbols

<Xt> Is the 64-bit name of the general-purpose destination register, encoded in the “Rt” field.

<systemreg> Is a System register name, encoded in “o0:op1:CRn:CRm:op2”.

The System register names are defined in AArch64 System Register Descriptions.

<op0> Is an unsigned immediate, encoded in “o0”:

o0	<op0>
0	2
1	3

<op1> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the “op1” field.

<Cn> Is a name ’Cn’, with ’n’ in the range 0 to 15, encoded in the “CRn” field.

<Cm> Is a name ’Cm’, with ’m’ in the range 0 to 15, encoded in the “CRm” field.

<op2> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the “op2” field.

Operation

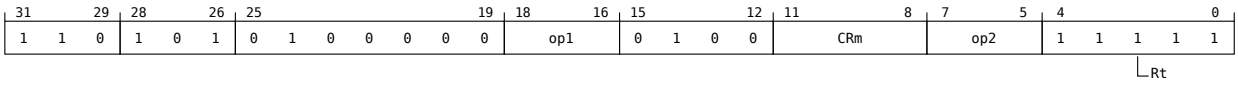
```
AArch64.SysRegRead(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2, t);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.184 MSR (immediate)

Move immediate value to special register

This instruction moves an immediate value to selected bits of the PSTATE. For more information, see Process state, PSTATE.



Encoding Applies when `(!(op1 == 000 && op2 IN {00x, 010}))`

```
MSR <pstatefield>, #<imm>
```

Decode for this encoding

```
if op1 == '000' && op2 == '000' then SEE "CFINV";

bits(2) min_EL;
boolean need_secure = FALSE;

case op1 of
    when '00x'
        min_EL = EL1;
    when '010'
        min_EL = EL1;
    when '011'
        min_EL = EL0;
    when '100'
        min_EL = EL2;
    when '101'
        EndOfDecode(Decode_UNDEF);
    when '110'
        min_EL = EL3;
    when '111'
        min_EL = EL1;
        need_secure = TRUE;

constant bits(4) operand = CRm;
PSTATEField field;
case op1:op2 of
    when '000 011'
        if !IsFeatureImplemented(FEAT_UAO) then EndOfDecode(Decode_UNDEF);
        field = PSTATEField_UAO;
    when '000 100'
        if !IsFeatureImplemented(FEAT_PAN) then EndOfDecode(Decode_UNDEF);
        field = PSTATEField_PAN;
    when '000 101' field = PSTATEField_SP;
    when '001 000'
        case CRm of
            otherwise
                EndOfDecode(Decode_UNDEF);
    when '011 010'
        if !IsFeatureImplemented(FEAT_DIT) then EndOfDecode(Decode_UNDEF);
        field = PSTATEField_DIT;
    when '011 110' field = PSTATEField_DAIFSet;
    when '011 111' field = PSTATEField_DAIFClr;
    when '011 001'
        if !IsFeatureImplemented(FEAT_SSBS) then EndOfDecode(Decode_UNDEF);
        field = PSTATEField_SSBS;
    otherwise      EndOfDecode(Decode_UNDEF);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<**pstatefield**> Is a PSTATE field name. For the MSR instruction, this is encoded in "op1:op2:op2":

op1	op2	<pstatefield>	Architectural Feature
000	011	UAO	FEAT_UAO
000	100	PAN	FEAT_PAN
000	101	SPSel	-
000	11x	RESERVED	-
001	xxx	RESERVED	-
010	xxx	RESERVED	-
011	000	RESERVED	-
011	001	SSBS	FEAT_SSBS
011	010	DIT	FEAT_DIT
011	011	RESERVED	-
011	10x	RESERVED	-
011	110	DAIFSet	-
011	111	DAIFClr	-
1xx	xxx	RESERVED	-

<**imm**> Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "CRm" field.

Operation

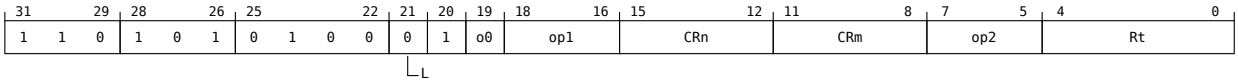
```
if UInt(PSTATE.EL) < UInt(min_EL) then UNDEFINED;
if need_secure && CurrentSecurityState() != SS_Secure then UNDEFINED;

case field of
    when PSTATEField_SSBS
        PSTATE.SSBS = operand<0>;
    when PSTATEField_SP
        PSTATE.SP = operand<0>;
    when PSTATEField_DAIFSet
        AArch64.CheckDAIFAccess(PSTATEField_DAIFSet);
        PSTATE.D = PSTATE.D OR operand<3>;
        PSTATE.A = PSTATE.A OR operand<2>;
        PSTATE.I = PSTATE.I OR operand<1>;
        PSTATE.F = PSTATE.F OR operand<0>;
    when PSTATEField_DAIFClr
        AArch64.CheckDAIFAccess(PSTATEField_DAIFClr);
        PSTATE.D = PSTATE.D AND NOT(operand<3>);
        PSTATE.A = PSTATE.A AND NOT(operand<2>);
        PSTATE.I = PSTATE.I AND NOT(operand<1>);
        PSTATE.F = PSTATE.F AND NOT(operand<0>);
    when PSTATEField_PAN
        PSTATE.PAN = operand<0>;
    when PSTATEField_UAO
        PSTATE.UAO = operand<0>;
    when PSTATEField_DIT
        PSTATE.DIT = operand<0>;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.185 MSR (register)

Move general-purpose register to System register

This instruction allows the PE to write an AArch64 System register from a general-purpose register.



Encoding

```
MSR (<systemreg>|S<op0>_<op1>_<Cn>_<Cm>_<op2>), <Xt>
```

Decode for this encoding

```
constant integer t       = UInt(Rt);
constant bits(1) sys_L   = L;
constant bits(2) sys_op0 = '1' : o0;
constant bits(3) sys_op1 = op1;
constant bits(3) sys_op2 = op2;
constant bits(4) sys_crn = CRn;
constant bits(4) sys_crm = CRm;
```

Assembler Symbols

<**systemreg**> Is a System register name, encoded in “o0:op1:CRn:CRm:op2”.

The System register names are defined in AArch64 System Register Descriptions.

<**op0**> Is an unsigned immediate, encoded in “o0”:

o0	<op0>
0	2
1	3

<**op1**> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the “op1” field.

<**Cn**> Is a name ’Cn’, with ’n’ in the range 0 to 15, encoded in the “CRn” field.

<**Cm**> Is a name ’Cm’, with ’m’ in the range 0 to 15, encoded in the “CRm” field.

<**op2**> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the “op2” field.

<**Xt**> Is the 64-bit name of the general-purpose source register, encoded in the “Rt” field.

Operation

```
AArch64.SysRegWrite(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2, t);
```

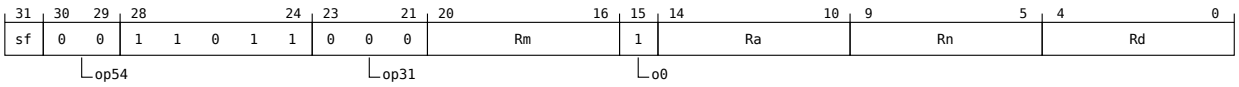

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.186 MSUB

Multiply-subtract

This instruction multiplies two register values, subtracts the product from a third register value, and writes the result to the destination register.

This instruction is used by the alias MNEG.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
MSUB <Wd>, <Wn>, <Wm>, <Wa>
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
MSUB <Xd>, <Xn>, <Xm>, <Xa>
```

Decode for all variants of this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer a = UInt(Ra);
constant integer datasize = 32 << UInt(sf);
```

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wn**> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<**Wm**> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

<**Wa**> Is the 32-bit name of the third general-purpose source register holding the minuend, encoded in the “Ra” field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Xn**> Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<**Xm**> Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

<**Xa**> Is the 64-bit name of the third general-purpose source register holding the minuend, encoded in the “Ra” field.

Alias Conditions

Alias	Is preferred when
MNEG	`Ra == '11111'`

Operation

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant bits(datasize) operand1 = X[n, datasize];
constant bits(datasize) operand2 = X[m, datasize];
constant bits(datasize) operand3 = X[a, datasize];

constant integer result = UInt(operand3) - (UInt(operand1) * UInt(operand2));

X[d, datasize] = result<datasize-1:0>;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

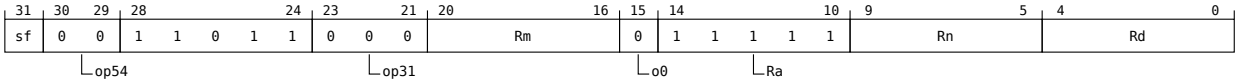
C6.2.187 MUL

Multiply

This instruction multiplies two register values and writes the result to the destination register.

This is an alias of MADD. This means:

- The encodings in this description are named to match the encodings of MADD.
- The description of MADD gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
MUL <Wd>, <Wn>, <Wm>
```

is equivalent to

```
MADD <Wd>, <Wn>, <Wm>, WZR
```

and is always the preferred disassembly.

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
MUL <Xd>, <Xn>, <Xm>
```

is equivalent to

```
MADD <Xd>, <Xn>, <Xm>, XZR
```

and is always the preferred disassembly.

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wn**> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<**Wm**> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Xn**> Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<**Xm**> Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

Operation

The description of MADD gives the operational pseudocode for this instruction.

Operational Information

The description of MADD gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

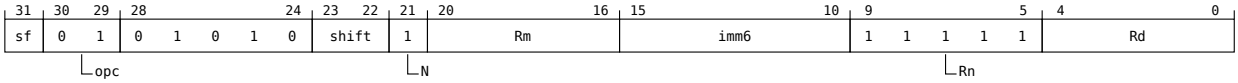
C6.2.188 MVN

Bitwise NOT

This instruction writes the bitwise inverse of a register value to the destination register.

This is an alias of ORN (shifted register). This means:

- The encodings in this description are named to match the encodings of ORN (shifted register).
- The description of ORN (shifted register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
MVN <Wd>, <Wm>{, <shift> #<amount>}
```

is equivalent to

```
ORN <Wd>, WZR, <Wm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
MVN <Xd>, <Xm>{, <shift> #<amount>}
```

is equivalent to

```
ORN <Xd>, XZR, <Xm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Wm**> Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.

<**shift**> Is the optional shift to be applied to the final source, defaulting to LSL and encoded in "shift":

shift	<**shift**>
00	LSL
01	LSR
10	ASR
11	ROR

<**amount**> For the "32-bit" variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the "64-bit" variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**Xm**> Is the 64-bit name of the general-purpose source register, encoded in the “Rm” field.

Operation

The description of ORN (shifted register) gives the operational pseudocode for this instruction.

Operational Information

The description of ORN (shifted register) gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

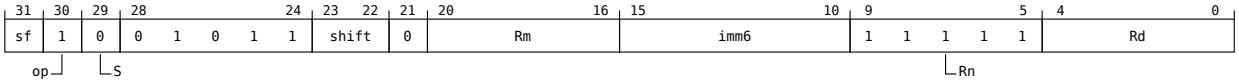
C6.2.189 NEG (shifted register)

Negate (shifted register)

This instruction negates an optionally-shifted register value, and writes the result to the destination register.

This is an alias of SUB (shifted register). This means:

- The encodings in this description are named to match the encodings of SUB (shifted register).
- The description of SUB (shifted register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
NEG <Wd>, <Wm>{, <shift> #<amount>}
```

is equivalent to

```
SUB <Wd>, WZR, <Wm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
NEG <Xd>, <Xm>{, <shift> #<amount>}
```

is equivalent to

```
SUB <Xd>, XZR, <Xm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Wm**> Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.

<**shift**> Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in "shift":

shift	<**shift**>
00	LSL
01	LSR
10	ASR
11	RESERVED

<**amount**> For the "32-bit" variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the "64-bit" variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xm> Is the 64-bit name of the general-purpose source register, encoded in the “Rm” field.

Operation

The description of SUB (shifted register) gives the operational pseudocode for this instruction.

Operational Information

The description of SUB (shifted register) gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

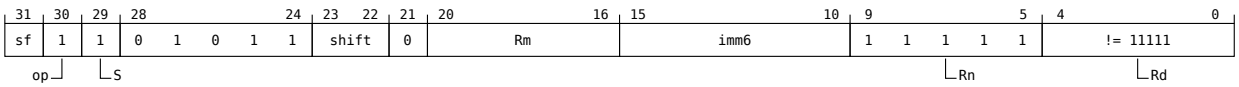
C6.2.190 NEGS

Negate, setting flags

This instruction negates an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

This is an alias of SUBS (shifted register). This means:

- The encodings in this description are named to match the encodings of SUBS (shifted register).
- The description of SUBS (shifted register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
NEGS <Wd>, <Wm>{, <shift> #<amount>}
```

is equivalent to

```
SUBS <Wd>, WZR, <Wm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
NEGS <Xd>, <Xm>{, <shift> #<amount>}
```

is equivalent to

```
SUBS <Xd>, XZR, <Xm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wm**> Is the 32-bit name of the general-purpose source register, encoded in the “Rm” field.

<**shift**> Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in “shift”:

shift	<**shift**>
00	LSL
01	LSR
10	ASR
11	RESERVED

<**amount**> For the “32-bit” variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm6” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xm> Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.

Operation

The description of SUBS (shifted register) gives the operational pseudocode for this instruction.

Operational Information

The description of SUBS (shifted register) gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

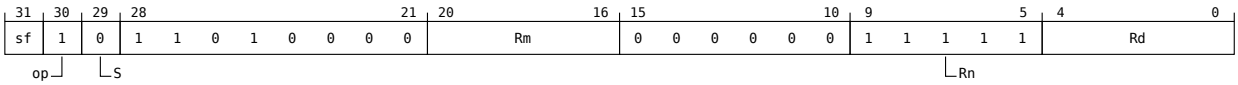
C6.2.191 NGC

Negate with carry

This instruction negates the sum of a register value and the value of NOT (Carry flag), and writes the result to the destination register.

This is an alias of SBC. This means:

- The encodings in this description are named to match the encodings of SBC.
- The description of SBC gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
NGC <Wd>, <Wm>
```

is equivalent to

```
SBC <Wd>, WZR, <Wm>
```

and is always the preferred disassembly.

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
NGC <Xd>, <Xm>
```

is equivalent to

```
SBC <Xd>, XZR, <Xm>
```

and is always the preferred disassembly.

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wm**> Is the 32-bit name of the general-purpose source register, encoded in the “Rm” field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Xm**> Is the 64-bit name of the general-purpose source register, encoded in the “Rm” field.

Operation

The description of SBC gives the operational pseudocode for this instruction.

Operational Information

The description of SBC gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

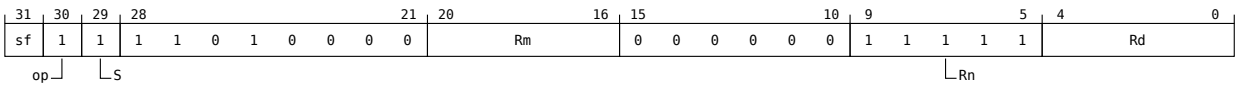
C6.2.192 NGCS

Negate with carry, setting flags

This instruction negates the sum of a register value and the value of NOT (Carry flag), and writes the result to the destination register. It updates the condition flags based on the result.

This is an alias of SBCS. This means:

- The encodings in this description are named to match the encodings of SBCS.
- The description of SBCS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
NGCS <Wd>, <Wm>
```

is equivalent to

```
SBCS <Wd>, WZR, <Wm>
```

and is always the preferred disassembly.

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
NGCS <Xd>, <Xm>
```

is equivalent to

```
SBCS <Xd>, XZR, <Xm>
```

and is always the preferred disassembly.

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wm**> Is the 32-bit name of the general-purpose source register, encoded in the “Rm” field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Xm**> Is the 64-bit name of the general-purpose source register, encoded in the “Rm” field.

Operation

The description of SBCS gives the operational pseudocode for this instruction.

Operational Information

The description of SBCS gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

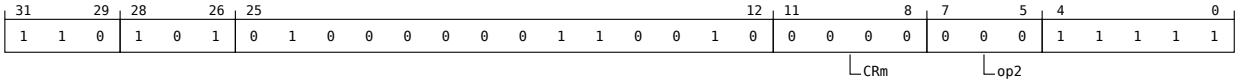
C6.2.193 NOP

No operation

This instruction does nothing, other than advance the value of the program counter by 4. This instruction can be used for instruction alignment purposes.

Note

The timing effects of including a NOP instruction in a program are not guaranteed. It can increase execution time, leave it unchanged, or even reduce it. Therefore, NOP instructions are not suitable for timing loops.



Encoding

NOP

Decode for this encoding

```
// Empty.
```

Operation

```
return; // Do nothing
```

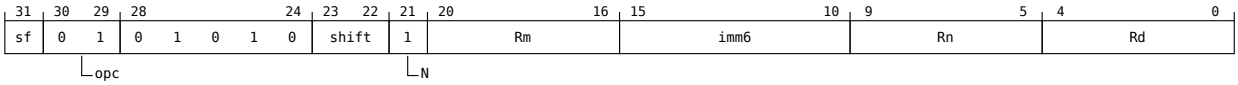

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.194 ORN (shifted register)

Bitwise OR NOT (shifted register)

This instruction performs a bitwise (inclusive) OR of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register.

This instruction is used by the alias MVN.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
ORN <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
ORN <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

Decode for all variants of this encoding

```
if sf == '0' && imm6<5> == '1' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
constant ShiftType shift_type = DecodeShift(shift);
constant integer shift_amount = UInt(imm6);
```

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wn**> Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Wm**> Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<**shift**> Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:

shift	<**shift**>
00	LSL
01	LSR
10	ASR
11	ROR

<**amount**> For the “32-bit” variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm6” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Xn**> Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Xm**> Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias Conditions

Alias	Is preferred when
MVN	Rn == '11111'

Operation

```
constant bits(datasize) operand1 = X[n, datasize];
constant bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount, datasize);

X[d, datasize] = operand1 OR NOT(operand2);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

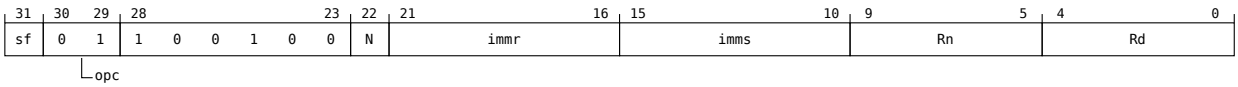
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.195 ORR (immediate)

Bitwise OR (immediate)

This instruction performs a bitwise (inclusive) OR of a register value and an immediate value, and writes the result to the destination register.

This instruction is used by the alias MOV (bitmask immediate).



Encoding for the 32-bit variant Applies when `(sf == 0 && N == 0)`

```
ORR <Wd|WSP>, <Wn>, #<imm>
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
ORR <Xd|SP>, <Xn>, #<imm>
```

Decode for all variants of this encoding

```
if sf == '0' && N != '0' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);
bits(datasize) imm;
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE, datasize);
```

Assembler Symbols

<**Wd|WSP**> Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

<**Wn**> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

<**imm**> For the "32-bit" variant: is the bitmask immediate, encoded in "imms:immr".

For the "64-bit" variant: is the bitmask immediate, encoded in "N:imms:immr".

<**Xd|SP**> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

<**Xn**> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
MOV (bitmask immediate)	`Rn == '11111' && !MoveWidePreferred(sf, N, imms, immr)`

Operation

```
constant bits(datasize) operand1 = X[n, datasize];
constant bits(datasize) operand2 = imm;

constant bits(datasize) result = operand1 OR operand2;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if d == 31 then
    SP[64] = ZeroExtend(result, 64);
else
    X[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

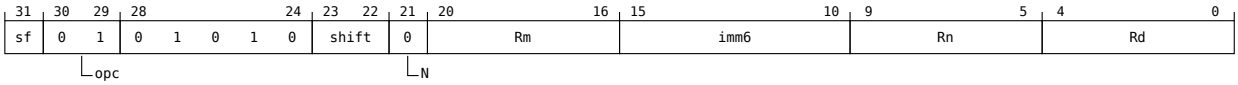
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.196 ORR (shifted register)

Bitwise OR (shifted register)

This instruction performs a bitwise (inclusive) OR of a register value and an optionally-shifted register value, and writes the result to the destination register.

This instruction is used by the alias MOV (register).



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
ORR <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
ORR <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

Decode for all variants of this encoding

```
if sf == '0' && imm6<5> == '1' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
constant ShiftType shift_type = DecodeShift(shift);
constant integer shift_amount = UInt(imm6);
```

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wn**> Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Wm**> Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<**shift**> Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:

shift	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<**amount**> For the “32-bit” variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm6” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Xn**> Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Xm**> Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias Conditions

Alias	Is preferred when
MOV (register)	shift == '00' && imm6 == '000000' && Rn == '11111'

Operation

```
constant bits(datasize) operand1 = X[n, datasize];
constant bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount, datasize);

X[d, datasize] = operand1 OR operand2;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.197 PACDA, PACDZA

Pointer Authentication Code for data address, using key A

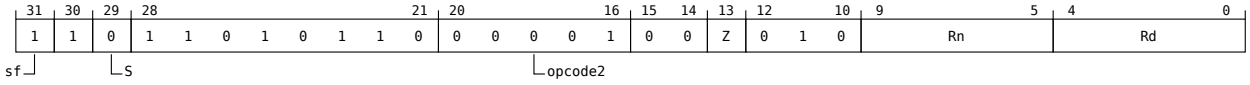
This instruction calculates and inserts a Pointer Authentication Code for a data address, using a modifier and key A.

The address is in the general-purpose register that is specified by <Xd>.

The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xn|SP> for PACDA.
- The value zero, for PACDZA.

Integer
(FEAT_PAuth)



Encoding for the PACDA variant Applies when (Z == 0)

```
PACDA <Xd>, <Xn|SP>
```

Encoding for the PACDZA variant Applies when (Z == 1 && Rn == 11111)

```
PACDZA <Xd>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_UNDEF);

boolean source_is_sp = FALSE;
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

if Z == '0' then // PACDA
    if n == 31 then source_is_sp = TRUE;
else // PACDZA
    if n != 31 then EndOfDecode(Decode_UNDEF);
```

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn|SP> Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

Operation

```
if source_is_sp then
    X[d, 64] = AddPACDA(X[d, 64], SP[64]);
else
    X[d, 64] = AddPACDA(X[d, 64], X[n, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.198 PACDB, PACDZB

Pointer Authentication Code for data address, using key B

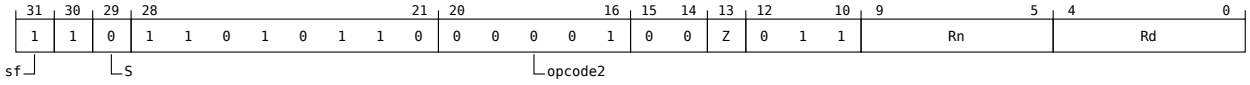
This instruction calculates and inserts a Pointer Authentication Code for a data address, using a modifier and key B.

The address is in the general-purpose register that is specified by <Xd>.

The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xn|SP> for PACDB.
- The value zero, for PACDZB.

Integer
(FEAT_PAuth)



Encoding for the PACDB variant Applies when (Z == 0)

```
PACDB <Xd>, <Xn|SP>
```

Encoding for the PACDZB variant Applies when (Z == 1 && Rn == 11111)

```
PACDZB <Xd>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_UNDEF);

boolean source_is_sp = FALSE;
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

if Z == '0' then // PACDB
    if n == 31 then source_is_sp = TRUE;
else // PACDZB
    if n != 31 then EndOfDecode(Decode_UNDEF);
```

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn|SP> Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

Operation

```
if source_is_sp then
    X[d, 64] = AddPACDB(X[d, 64], SP[64]);
else
    X[d, 64] = AddPACDB(X[d, 64], X[n, 64]);
```

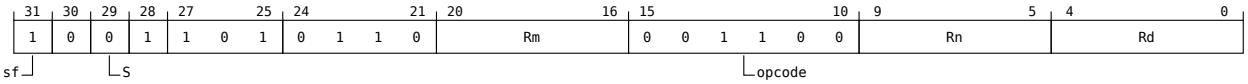

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.199 PACGA

Pointer Authentication Code, using generic key

This instruction calculates the Pointer Authentication Code for a 64-bit value in the first source register, using a modifier in the second source register, and the generic key. The calculated Pointer Authentication Code is written to the most significant 32 bits of the destination register, and the least significant 32 bits of the destination register are set to zero.

Integer
(FEAT_PAuth)



Encoding

```
PACGA <Xd>, <Xn>, <Xm|SP>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_UNDEF);

boolean source_is_sp = FALSE;
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

if m == 31 then source_is_sp = TRUE;
```

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

<Xm|SP> Is the 64-bit name of the second general-purpose source register or stack pointer, encoded in the "Rm" field.

Operation

```
if source_is_sp then
    X[d, 64] = AddPACGA(X[n, 64], SP[64]);
else
    X[d, 64] = AddPACGA(X[n, 64], X[m, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.200 PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA

Pointer Authentication Code for instruction address, using key A

This instruction calculates and inserts a Pointer Authentication Code for an instruction address, using a modifier and key A.

The address is:

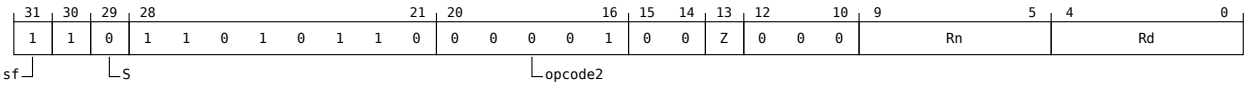
- In the general-purpose register that is specified by <Xd> for PACIA and PACIZA.
- In X17, for PACIA1716.
- In X30, for PACIASP and PACIAZ.

The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xn|SP> for PACIA.
- The value zero, for PACIZA and PACIAZ.
- In X16, for PACIA1716.
- In SP, for PACIASP.

It has encodings from 2 classes: Integer and System

Integer
(FEAT_PAuth)



Encoding for the PACIA variant Applies when (Z == 0)

```
PACIA <Xd>, <Xn|SP>
```

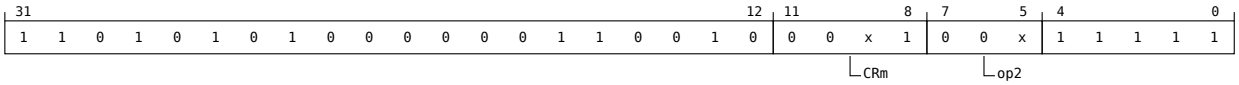
Encoding for the PACIZA variant Applies when (Z == 1 && Rn == 11111)

```
PACIZA <Xd>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_UNDEF);
boolean source_is_sp = FALSE;
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

if Z == '0' then // PACIA
    if n == 31 then source_is_sp = TRUE;
else // PACIZA
    if n != 31 then EndOfDecode(Decode_UNDEF);
```

System
(FEAT_PAuth)



Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the PACIA1716 variant Applies when `(CRm == 0001 && op2 == 000)`

```
PACIA1716
```

Encoding for the PACIASP variant Applies when `(CRm == 0011 && op2 == 001)`

```
PACIASP
```

Encoding for the PACIAZ variant Applies when `(CRm == 0011 && op2 == 000)`

```
PACIAZ
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_NOP);
integer d;
integer n;
boolean source_is_sp = FALSE;

case CRm:op2 of
    when '0011 000' // PACIAZ
        d = 30;
        n = 31;
    when '0011 001' // PACIASP
        d = 30;
        source_is_sp = TRUE;
    when '0001 000' // PACIA1716
        d = 17;
        n = 16;
```

Assembler Symbols

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Xn**|**SP**> Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the “Rn” field.

Operation

```
if source_is_sp then
    X[d, 64] = AddPACIA(X[d, 64], SP[64]);
else
    X[d, 64] = AddPACIA(X[d, 64], X[n, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.201 PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB

Pointer Authentication Code for instruction address, using key B

This instruction calculates and inserts a Pointer Authentication Code for an instruction address, using a modifier and key B.

The address is:

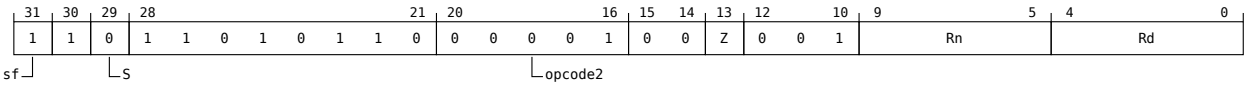
- In the general-purpose register that is specified by <Xd> for PACIB and PACIZB.
- In X17, for PACIB1716.
- In X30, for PACIBSP and PACIBZ.

The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xn|SP> for PACIB.
- The value zero, for PACIZB and PACIBZ.
- In X16, for PACIB1716.
- In SP, for PACIBSP.

It has encodings from 2 classes: Integer and System

Integer
(FEAT_PAuth)



Encoding for the PACIB variant Applies when (Z == 0)

```
PACIB <Xd>, <Xn|SP>
```

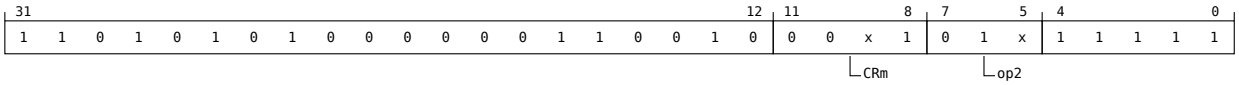
Encoding for the PACIZB variant Applies when (Z == 1 && Rn == 11111)

```
PACIZB <Xd>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_UNDEF);
boolean source_is_sp = FALSE;
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

if Z == '0' then // PACIB
    if n == 31 then source_is_sp = TRUE;
else // PACIZB
    if n != 31 then EndOfDecode(Decode_UNDEF);
```

System
(FEAT_PAuth)



Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the PACIB1716 variant Applies when `(CRm == 0001 && op2 == 010)`

```
PACIB1716
```

Encoding for the PACIBSP variant Applies when `(CRm == 0011 && op2 == 011)`

```
PACIBSP
```

Encoding for the PACIBZ variant Applies when `(CRm == 0011 && op2 == 010)`

```
PACIBZ
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_NOP);
integer d;
integer n;
boolean source_is_sp = FALSE;

case CRm:op2 of
    when '0011 010' // PACIBZ
        d = 30;
        n = 31;
    when '0011 011' // PACIBSP
        d = 30;
        source_is_sp = TRUE;
    when '0001 010' // PACIB1716
        d = 17;
        n = 16;
```

Assembler Symbols

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Xn**|**SP**> Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the “Rn” field.

Operation

```
if source_is_sp then
    X[d, 64] = AddPACIB(X[d, 64], SP[64]);
else
    X[d, 64] = AddPACIB(X[d, 64], X[n, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.202 PRFM (immediate)

Prefetch memory (immediate)

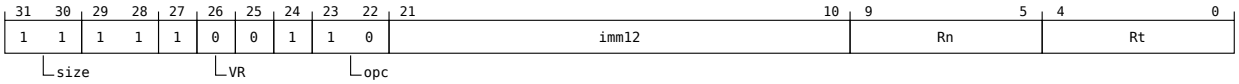
This instruction signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The address for data memory accesses is calculated from a base register value and an immediate offset. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as making the cache line containing the specified address available at the level of cache specified by the instruction.

The <prfop> operand specifies the prefetch hint as follows:

- Access type:
  - PLD for prefetch for load.
  - PLI for prefetch for execute.
  - PST for prefetch for store.
- Target cache level:
  - L1 for Level 1 cache.
  - L2 for Level 2 cache.
  - L3 for Level 3 cache.
- Policy:
  - KEEP for retained or temporal prefetch, allocated in the cache normally.
  - STRM for streaming or non-temporal prefetch, for data that is used only once.

The effect of a `PRFM` instruction is IMPLEMENTATION DEFINED. For more information, see Prefetch memory.

For information about addressing modes, see Load/Store addressing modes.



Encoding

```
PRFM (<prfop>|#<imm5>), [<Xn|SP>{, #<pimm>}]
```

Decode for this encoding

```
constant bits(64) offset = LSL(ZeroExtend(imm12, 64), 3);
constant integer n = UInt(Rn);
constant integer t = UInt(Rt);
constant boolean nontemporal = FALSE;
```

Assembler Symbols

<prfop> Is the prefetch operation, encoded in "Rt":

Rt	<prfop>
00000	PLDL1KEEP
00001	PLDL1STRM
00010	PLDL2KEEP
00011	PLDL2STRM
00100	PLDL3KEEP

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Rt	<prfop>
00101	PLDL3STRM
01000	PLIL1KEEP
01001	PLIL1STRM
01010	PLIL2KEEP
01011	PLIL2STRM
01100	PLIL3KEEP
01101	PLIL3STRM
10000	PSTL1KEEP
10001	PSTL1STRM
10010	PSTL2KEEP
10011	PSTL2STRM
10100	PSTL3KEEP
10101	PSTL3STRM

For other encodings of the “Rt” field, use <imm5>.

<**imm5**> Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the “Rt” field.

This syntax is only for encodings that are not accessible using <prfop>.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<**pimm**> Is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the “imm12” field as <pimm>/8.

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_PREFETCH, nontemporal, privileged, t);

if n == 31 then
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

Prefetch(address, t<4:0>);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.203 PRFM (literal)

Prefetch memory (literal)

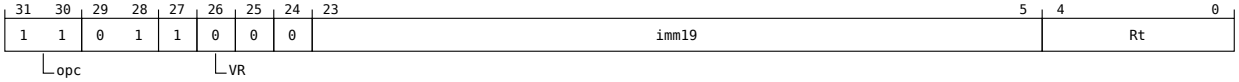
This instruction signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The address for data memory accesses is calculated from the PC value and an immediate offset. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as making the cache line containing the specified address available at the level of cache specified by the instruction.

The <prfop> operand specifies the prefetch hint as follows:

- Access type:
  - PLD for prefetch for load.
  - PLI for prefetch for execute.
  - PST for prefetch for store.
- Target cache level:
  - L1 for Level 1 cache.
  - L2 for Level 2 cache.
  - L3 for Level 3 cache.
- Policy:
  - KEEP for retained or temporal prefetch, allocated in the cache normally.
  - STRM for streaming or non-temporal prefetch, for data that is used only once.

The effect of a `PRFM` instruction is IMPLEMENTATION DEFINED. For more information, see Prefetch memory.

For information about addressing modes, see Load/Store addressing modes.



Encoding

```
PRFM (<prfop>|#<imm5>), <label>
```

Decode for this encoding

```
constant integer t = UInt(Rt);

constant bits(64) offset = SignExtend(imm19:'00', 64);
```

Assembler Symbols

<prfop> Is the prefetch operation, encoded in "Rt":

Rt	<prfop>
00000	PLDL1KEEP
00001	PLDL1STRM
00010	PLDL2KEEP
00011	PLDL2STRM
00100	PLDL3KEEP
00101	PLDL3STRM

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Rt	<prfop>
01000	PLIL1KEEP
01001	PLIL1STRM
01010	PLIL2KEEP
01011	PLIL2STRM
01100	PLIL3KEEP
01101	PLIL3STRM
10000	PSTL1KEEP
10001	PSTL1STRM
10010	PSTL2KEEP
10011	PSTL2STRM
10100	PSTL3KEEP
10101	PSTL3STRM

For other encodings of the “Rt” field, use <imm5>.

<**imm5**> Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the “Rt” field.

This syntax is only for encodings that are not accessible using <prfop>.

<**label**> Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as “imm19” times 4.

Operation

```
constant bits(64) address = PC64 + offset;

Prefetch(address, t<4:0>);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.204 PRFM (register)

Prefetch memory (register)

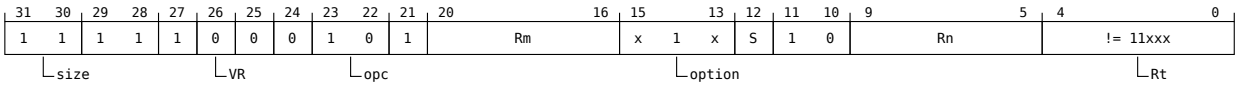
This instruction signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The address for data memory accesses is calculated from a base register value and an offset register value. The offset register value can optionally be shifted and extended. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as making the cache line containing the specified address available at the level of cache specified by the instruction.

The <prfop> operand specifies the prefetch hint as follows:

- Access type:
  - PLD for prefetch for load.
  - PLI for prefetch for execute.
  - PST for prefetch for store.
- Target cache level:
  - L1 for Level 1 cache.
  - L2 for Level 2 cache.
  - L3 for Level 3 cache.
- Policy:
  - KEEP for retained or temporal prefetch, allocated in the cache normally.
  - STRM for streaming or non-temporal prefetch, for data that is used only once.

The effect of a PRFM instruction is IMPLEMENTATION DEFINED. For more information, see Prefetch memory.

For information about addressing modes, see Load/Store addressing modes.



Encoding

```
PRFM (<prfop>|#<imm5>), [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Decode for this encoding

```
if option<1> == '0' then EndOfDecode(Decode_UNDEF);             // sub-word index
constant ExtendType extend_type = DecodeRegExtend(option);
constant integer shift = if S == '1' then 3 else 0;
constant integer n = UInt(Rn);
constant integer t = UInt(Rt);
constant integer m = UInt(Rm);
constant boolean nontemporal = FALSE;
```

Assembler Symbols

<prfop> Is the prefetch operation, encoded in "Rt":

Rt	<prfop>
00000	PLDL1KEEP
00001	PLDL1STRM
00010	PLDL2KEEP

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Rt	<prfop>
00011	PLDL2STRM
00100	PLDL3KEEP
00101	PLDL3STRM
01000	PLIL1KEEP
01001	PLIL1STRM
01010	PLIL2KEEP
01011	PLIL2STRM
01100	PLIL3KEEP
01101	PLIL3STRM
10000	PSTL1KEEP
10001	PSTL1STRM
10010	PSTL2KEEP
10011	PSTL2STRM
10100	PSTL3KEEP
10101	PSTL3STRM

For other encodings of the “Rt” field, use <imm5>.

<**imm5**> Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the “Rt” field.

This syntax is only for encodings that are not accessible using <prfop>.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<**Wm**> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the “Rm” field.

<**Xm**> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the “Rm” field.

<**extend**> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted, encoded in “option”:

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<**amount**> Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#3

Operation

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
bits(64) address;

constant bits(64) offset = ExtendReg(m, extend_type, shift, 64);
constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_PREFETCH, nontemporal, privileged, t);

if n == 31 then
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

Prefetch(address, t<4:0>);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.205 PRFUM

Prefetch memory (unscaled offset)

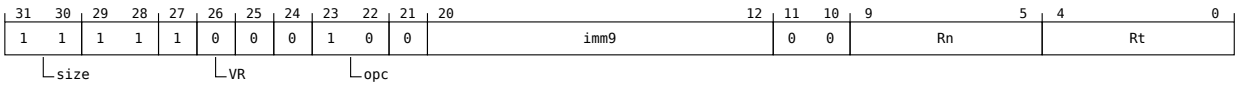
This instruction signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as making the cache line containing the specified address available at the level of cache specified by the instruction.

The <prfop> operand specifies the prefetch hint as follows:

- Access type:
  - PLD for prefetch for load.
  - PLI for prefetch for execute.
  - PST for prefetch for store.
  - L1 for Level 1 cache.
  - L2 for Level 2 cache.
  - L3 for Level 3 cache.
- Policy:
  - KEEP for retained or temporal prefetch, allocated in the cache normally.
  - STRM for streaming or non-temporal prefetch, for data that is used only once.

The effect of a `PRFUM` instruction is IMPLEMENTATION DEFINED. For more information, see Prefetch memory.

For information about addressing modes, see Load/Store addressing modes.



Encoding

```
PRFUM (<prfop>|#<imm5>), [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
constant bits(64) offset = SignExtend(imm9, 64);
constant integer n = UInt(Rn);
constant integer t = UInt(Rt);
constant boolean nontemporal = FALSE;
```

Assembler Symbols

<prfop> Is the prefetch operation, encoded in "Rt":

Rt	<prfop>
00000	PLDL1KEEP
00001	PLDL1STRM
00010	PLDL2KEEP
00011	PLDL2STRM
00100	PLDL3KEEP
00101	PLDL3STRM

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Rt	<prfop>
01000	PLIL1KEEP
01001	PLIL1STRM
01010	PLIL2KEEP
01011	PLIL2STRM
01100	PLIL3KEEP
01101	PLIL3STRM
10000	PSTL1KEEP
10001	PSTL1STRM
10010	PSTL2KEEP
10011	PSTL2STRM
10100	PSTL3KEEP
10101	PSTL3STRM

For other encodings of the “Rt” field, use <imm5>.

<**imm5**> Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the “Rt” field.

This syntax is only for encodings that are not accessible using <prfop>.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<**simm**> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_PREFETCH, nontemporal, privileged, t);

if n == 31 then
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

Prefetch(address, t<4:0>);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

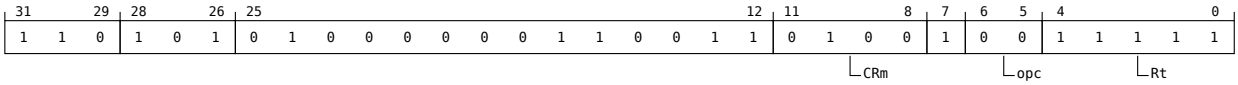
C6.2.206 PSSBB

Physical speculative store bypass barrier

This instruction is a memory barrier that prevents speculative loads from bypassing earlier stores to the same physical address under certain conditions. For more information and details of the semantics, see Physical Speculative Store Bypass Barrier (PSSBB).

This is an alias of DSB. This means:

- The encodings in this description are named to match the encodings of DSB.
- The description of DSB gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding

```
PSSBB
```

is equivalent to

```
DSB #4
```

and is always the preferred disassembly.

Operation

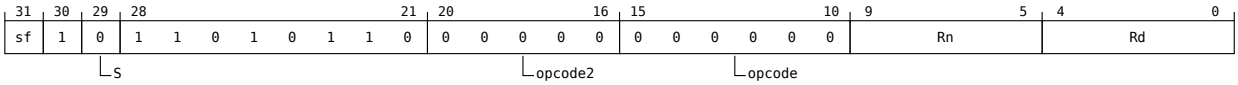
The description of DSB gives the operational pseudocode for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.207 RBIT

Reverse bits

This instruction reverses the bit order in a register.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
RBIT <Wd>, <Wn>
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
RBIT <Xd>, <Xn>
```

Decode for all variants of this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);
```

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wn**> Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Xn**> Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

```
constant bits(datasize) operand = X[n, datasize];
bits(datasize) result;

for i = 0 to datasize-1
    result<(datasize-1)-i> = operand<i>;

X[d, datasize] = result;
```

Operational information

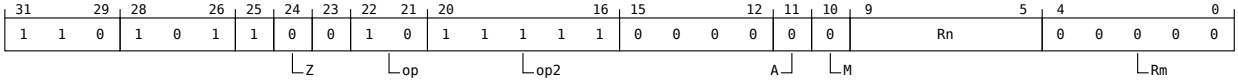
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.208 RET

Return from subroutine

This instruction branches unconditionally to an address in a register. This instruction provides a hint that this is a subroutine return.



Encoding

```
RET {<Xn>}
```

Decode for this encoding

```
constant integer n = UInt(Rn);
```

Assembler Symbols

<Xn> Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field. Defaults to X30 if absent.

Operation

```
constant bits(64) target = X[n, 64];

constant boolean branch_conditional = FALSE;
BranchTo(target, BranchType_RET, branch_conditional);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.209 RETAA, RETAB

Return from subroutine, with pointer authentication

This instruction authenticates the address that is held in LR, using SP as the modifier and the specified key, and branches to the authenticated address.

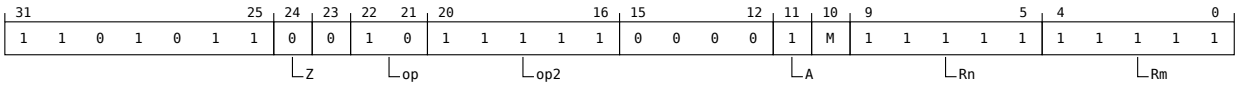
Key A is used for `RETAA`. Key B is used for `RETAB`.

If the authentication passes, the PE continues execution at the target of the branch. For information on behavior if the authentication fails, see Faulting on pointer authentication.

The authenticated address is not written back to LR.

This instruction provides a hint that this is a subroutine return.

Integer
(FEAT_PAuth)



Encoding for the RETAA variant Applies when `(M == 0)`

`RETAA`

Encoding for the RETAB variant Applies when `(M == 1)`

`RETAB`

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_UNDEF);

constant boolean use_key_a = (M == '0');
constant boolean auth_then_branch = TRUE;
```

Operation

```
bits(64) target = X[30, 64];

constant bits(64) modifier = SP[64];

if use_key_a then
    target = AuthIA(target, modifier, auth_then_branch);
else
    target = AuthIB(target, modifier, auth_then_branch);

constant boolean branch_conditional = FALSE;
BranchTo(target, BranchType_RET, branch_conditional);
```

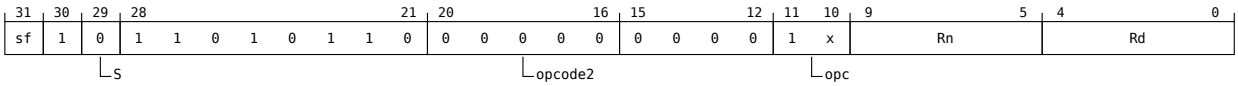

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.210 REV

Reverse bytes

This instruction reverses the byte order in a register.

This instruction is used by the pseudo-instruction REV64.



Encoding for the 32-bit variant Applies when `(sf == 0 && opc == 10)`

```
REV <Wd>, <Wn>
```

Encoding for the 64-bit variant Applies when `(sf == 1 && opc == 11)`

```
REV <Xd>, <Xn>
```

Decode for all variants of this encoding

```
if opc == '11' && sf == '0' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);
constant integer container_size = 8 << UInt(opc);
```

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Wn**> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Xn**> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

```
constant bits(datasize) operand = X[n, datasize];
bits(datasize) result;

constant integer containers = datasize DIV container_size;
for c = 0 to containers-1
    constant bits(container_size) container = Elem[operand, c, container_size];
    Elem[result, c, container_size] = Reverse(container, 8);

X[d, datasize] = result;
```

Operational information

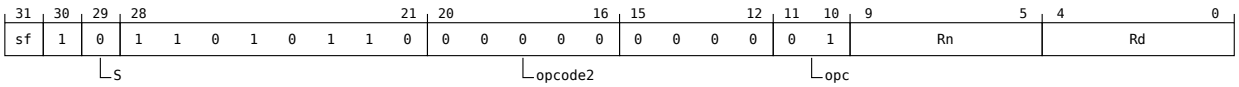
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.211 REV16

Reverse bytes in 16-bit halfwords

This instruction reverses the byte order in each 16-bit halfword of a register.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
REV16 <Wd>, <Wn>
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
REV16 <Xd>, <Xn>
```

Decode for all variants of this encoding

```
if opc == '11' && sf == '0' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);
constant integer container_size = 8 << UInt(opc);
```

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wn**> Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Xn**> Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

```
constant bits(datasize) operand = X[n, datasize];
bits(datasize) result;

constant integer containers = datasize DIV container_size;
for c = 0 to containers-1
    constant bits(container_size) container = Elem[operand, c, container_size];
    Elem[result, c, container_size] = Reverse(container, 8);

X[d, datasize] = result;
```

Operational information

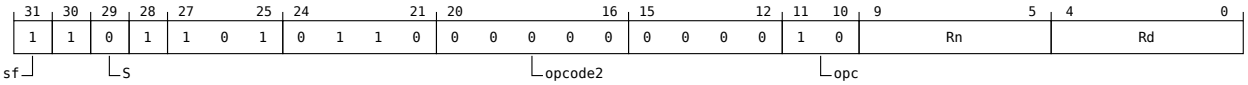
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.212 REV32

Reverse bytes in 32-bit words

This instruction reverses the byte order in each 32-bit word of a register.



Encoding

```
REV32 <Xd>, <Xn>
```

Decode for this encoding

```
if opc == '11' && sf == '0' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);
constant integer container_size = 8 << UInt(opc);
```

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

```
constant bits(datasize) operand = X[n, datasize];
bits(datasize) result;

constant integer containers = datasize DIV container_size;
for c = 0 to containers-1
    constant bits(container_size) container = Elem[operand, c, container_size];
    Elem[result, c, container_size] = Reverse(container, 8);

X[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

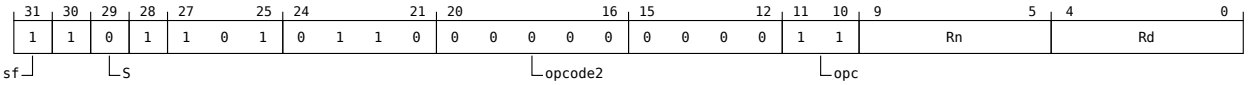
C6.2.213 REV64

Reverse bytes

This instruction reverses the byte order in a 64-bit general-purpose register.

This is a pseudo-instruction of REV. This means:

- The encodings in this description are named to match the encodings of REV.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of REV gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding

```
REV64 <Xd>, <Xn>
```

is equivalent to

```
REV <Xd>, <Xn>
```

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of REV gives the operational pseudocode for this instruction.

Operational Information

The description of REV gives the operational information for this instruction.

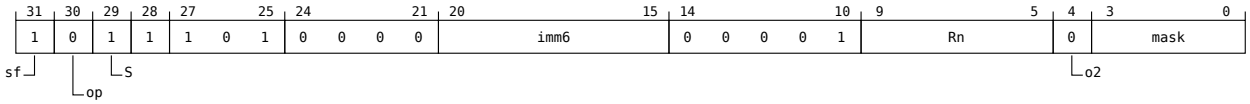
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.214 RMIF

Rotate, mask insert flags

This instruction performs a rotation right of a value held in a general-purpose register by an immediate value, and then inserts a selection of the bottom four bits of the result of the rotation into the PSTATE flags, under the control of a second immediate mask.

Integer
(FEAT_FlagM)



Encoding

```
RMIF <Xn>, #<shift>, #<mask>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FlagM) then EndOfDecode(Decode_UNDEF);
constant integer imm = UInt(imm6);
constant bits(4) flagmask = mask;
constant integer n = UInt(Rn);
```

Assembler Symbols

<**Xn**> Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

<**shift**> Is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

<**mask**> Is the flag bit mask, an immediate in the range 0 to 15, which selects the bits that are inserted into the NZCV condition flags, encoded in the “mask” field.

Operation

```
constant bits(64) reg = X[n, 64];
constant bits(4) flags = (reg:reg)<imm+3:imm>;
if flagmask<3> == '1' then PSTATE.N = flags<3>;
if flagmask<2> == '1' then PSTATE.Z = flags<2>;
if flagmask<1> == '1' then PSTATE.C = flags<1>;
if flagmask<0> == '1' then PSTATE.V = flags<0>;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

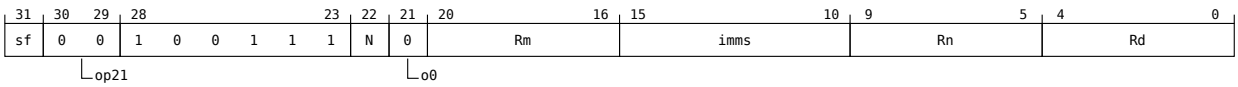
C6.2.215 ROR (immediate)

Rotate right (immediate)

This instruction provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left.

This is an alias of EXTR. This means:

- The encodings in this description are named to match the encodings of EXTR.
- The description of EXTR gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding for the 32-bit variant Applies when `(sf == 0 && N == 0 && imms == 0xxxxx)`

```
ROR <Wd>, <Ws>, #<shift>
```

is equivalent to

```
EXTR <Wd>, <Ws>, <Ws>, #<shift>
```

and is the preferred disassembly when `Rn == Rm`.

Encoding for the 64-bit variant Applies when `(sf == 1 && N == 1)`

```
ROR <Xd>, <Xs>, #<shift>
```

is equivalent to

```
EXTR <Xd>, <Xs>, <Xs>, #<shift>
```

and is the preferred disassembly when `Rn == Rm`.

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Ws**> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.

<**shift**> For the "32-bit" variant: is the amount by which to rotate, in the range 0 to 31, encoded in the "imms" field.

For the "64-bit" variant: is the amount by which to rotate, in the range 0 to 63, encoded in the "imms" field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Xs**> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.

Operation

The description of EXTR gives the operational pseudocode for this instruction.

Operational Information

The description of EXTR gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

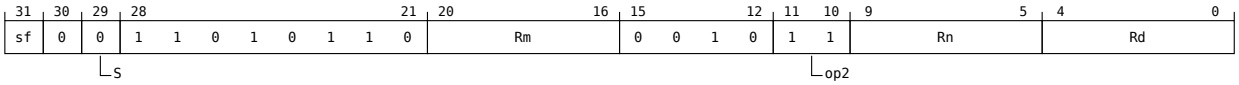
C6.2.216 ROR (register)

Rotate right (register)

This instruction provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. The value of the second source register modulo the register size in bits gives the number of bits by which the first source register is right-shifted.

This is an alias of RORV. This means:

- The encodings in this description are named to match the encodings of RORV.
- The description of RORV gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
ROR <Wd>, <Wn>, <Wm>
```

is equivalent to

```
RORV <Wd>, <Wn>, <Wm>
```

and is always the preferred disassembly.

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
ROR <Xd>, <Xn>, <Xm>
```

is equivalent to

```
RORV <Xd>, <Xn>, <Xm>
```

and is always the preferred disassembly.

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wn**> Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Wm**> Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the “Rm” field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Xn**> Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Xm**> Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the “Rm” field.

Operation

The description of RORV gives the operational pseudocode for this instruction.

Operational Information

The description of RORV gives the operational information for this instruction.

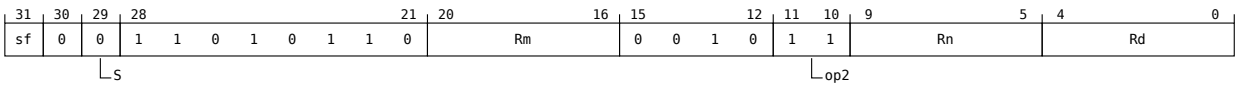
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.217 RORV

Rotate right variable

This instruction provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. The value of the second source register modulo the register size in bits gives the number of bits by which the first source register is right-shifted.

This instruction is used by the alias ROR (register).



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
RORV <Wd>, <Wn>, <Wm>
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
RORV <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
constant ShiftType shift_type = DecodeShift(op2);
```

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wn**> Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Wm**> Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the “Rm” field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Xn**> Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Xm**> Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the “Rm” field.

Operation

```
constant bits(datasize) operand2 = X[m, datasize];

X[d, datasize] = ShiftReg(n, shift_type, UInt(operand2) MOD datasize, datasize);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

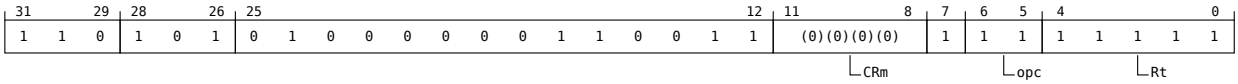
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.218 SB

Speculation barrier

This instruction is a barrier that controls speculation. For more information and details of the semantics, see Speculation Barrier (SB).

System
(FEAT_SB)



Encoding

SB

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SB) then EndOfDecode(Decode_UNDEF);
```

Operation

```
SpeculationBarrier();
```

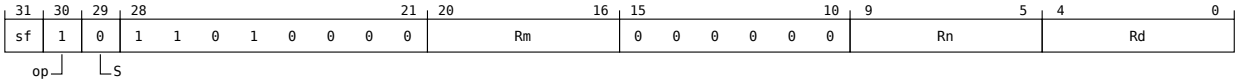

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.219 SBC

Subtract with carry

This instruction subtracts a register value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register.

This instruction is used by the alias NGC.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
SBC <Wd>, <Wn>, <Wm>
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
SBC <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
```

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wn**> Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Wm**> Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Xn**> Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Xm**> Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Alias Conditions

Alias	Is preferred when
NGC	`Rn == '11111'`

Operation

```
constant bits(datasize) operand1 = X[n, datasize];
constant bits(datasize) operand2 = NOT(X[m, datasize]);
bits(datasize) result;

(result, -) = AddWithCarry(operand1, operand2, PSTATE.C);

X[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

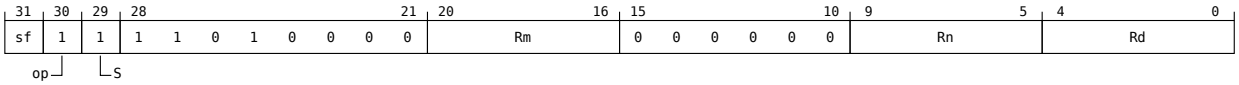
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.220 SBCS

Subtract with carry, setting flags

This instruction subtracts a register value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias NGCS.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
SBCS <Wd>, <Wn>, <Wm>
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
SBCS <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
```

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wn**> Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Wm**> Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Xn**> Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Xm**> Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Alias Conditions

Alias	Is preferred when
NGCS	`Rn == '11111'`

Operation

```
constant bits(datasize) operand1 = X[n, datasize];
constant bits(datasize) operand2 = NOT(X[m, datasize]);
bits(datasize) result;
bits(4) nzcv;

(result, nzcv) = AddWithCarry(operand1, operand2, PSTATE.C);

X[d, datasize] = result;
PSTATE.<N,Z,C,V> = nzcv;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

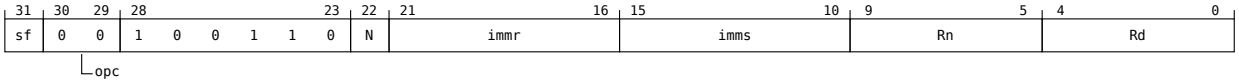
C6.2.221 SBFIZ

Signed bitfield insert in zeros

This instruction copies a bitfield of <width> bits from the least significant bits of the source register to bit position <lsb> of the destination register, setting the destination bits below the bitfield to zero, and the bits above the bitfield to a copy of the most significant bit of the bitfield.

This is an alias of SBFM. This means:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding for the 32-bit variant Applies when `(sf == 0 && N == 0)`

```
SBFIZ <Wd>, <Wn>, #<lsb>, #<width>
```

is equivalent to

```
SBFM <Wd>, <Wn>, #(-<lsb> MOD 32), #(<width>-1)
```

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

Encoding for the 64-bit variant Applies when `(sf == 1 && N == 1)`

```
SBFIZ <Xd>, <Xn>, #<lsb>, #<width>
```

is equivalent to

```
SBFM <Xd>, <Xn>, #(-<lsb> MOD 64), #(<width>-1)
```

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Wn**> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

<**lsb**> For the "32-bit" variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 31.

For the "64-bit" variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 63.

<**width**> For the "32-bit" variant: is the width of the bitfield, in the range 1 to 32-<lsb>.

For the "64-bit" variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Xn**> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Operational Information

The description of SBFM gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.222 SBFM

Signed bitfield move

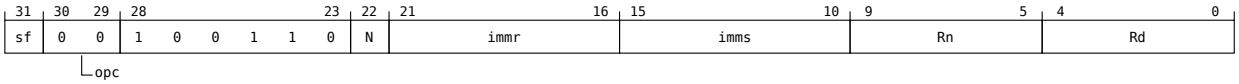
This instruction is usually accessed via one of its aliases, which are always preferred for disassembly.

If <imms> is greater than or equal to <immr>, this copies a bitfield of (<imms>-<immr>+1) bits starting from bit position <immr> in the source register to the least significant bits of the destination register.

If <imms> is less than <immr>, this copies a bitfield of (<imms>+1) bits from the least significant bits of the source register to bit position (regsize-<immr>) of the destination register, where regsize is the destination register size of 32 or 64 bits.

In both cases, the destination bits below the bitfield are set to zero, and the bits above the bitfield are set to a copy of the most significant bit of the bitfield.

This instruction is used by the aliases ASR (immediate), SBFIZ, SBFX, SXTB, SXTH, and SXTW.



Encoding for the 32-bit variant Applies when `(sf == 0 && N == 0)`

```
SBFM <Wd>, <Wn>, #<immr>, #<imms>
```

Encoding for the 64-bit variant Applies when `(sf == 1 && N == 1)`

```
SBFM <Xd>, <Xn>, #<immr>, #<imms>
```

Decode for all variants of this encoding

```
if sf == '1' && N != '1' then EndOfDecode(Decode_UNDEF);
if sf == '0' && (N != '0' || immr<5> != '0' || imms<5> != '0') then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);
constant integer s = UInt(imms);
constant integer r = UInt(immr);

bits(datasize) wmask;
bits(datasize) tmask;
(wmask, tmask) = DecodeBitMasks(N, imms, immr, FALSE, datasize);
```

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wn**> Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<**immr**> For the “32-bit” variant: is the right rotate amount, in the range 0 to 31, encoded in the “immr” field.

For the “64-bit” variant: is the right rotate amount, in the range 0 to 63, encoded in the “immr” field.

<**imms**> For the “32-bit” variant: is the leftmost bit number to be moved from the source, in the range 0 to 31, encoded in the “imms” field.

For the “64-bit” variant: is the leftmost bit number to be moved from the source, in the range 0 to 63, encoded in the “imms” field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Alias Conditions

Alias	Of variant	Is preferred when
ASR (immediate)	32-bit	imms == '011111'
ASR (immediate)	64-bit	imms == '111111'
SBFIZ		UInt(imms) < UInt(immr)
SBFX		BFXPreferred(sf, opc<1>, imms, immr)
SXTB		immr == '000000' && imms == '000111'
SXTH		immr == '000000' && imms == '001111'
SXTW		immr == '000000' && imms == '011111'

Operation

```
constant bits(datasize) src = X[n, datasize];

// Perform bitfield move on low bits
constant bits(datasize) bot = ROR(src, r) AND wmask;

constant bits(datasize) top = Replicate(src<s>, datasize);

// Combine extension bits and result bits
X[d, datasize] = (top AND NOT(tmask)) OR (bot AND tmask);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

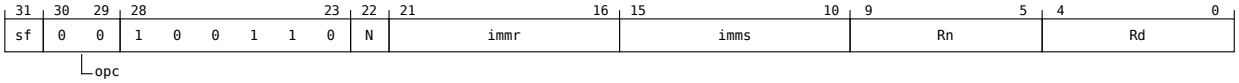
C6.2.223 SBFX

Signed bitfield extract

This instruction copies a bitfield of <width> bits starting from bit position <lsb> in the source register to the least significant bits of the destination register, and sets destination bits above the bitfield to a copy of the most significant bit of the bitfield.

This is an alias of SBFM. This means:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding for the 32-bit variant Applies when `(sf == 0 && N == 0)`

```
SBFX <Wd>, <Wn>, #<lsb>, #<width>
```

is equivalent to

```
SBFM <Wd>, <Wn>, #<lsb>, #(<lsb>+<width>-1)
```

and is the preferred disassembly when `BFXPreferred(sf, opc<1>, imms, immr)`.

Encoding for the 64-bit variant Applies when `(sf == 1 && N == 1)`

```
SBFX <Xd>, <Xn>, #<lsb>, #<width>
```

is equivalent to

```
SBFM <Xd>, <Xn>, #<lsb>, #(<lsb>+<width>-1)
```

and is the preferred disassembly when `BFXPreferred(sf, opc<1>, imms, immr)`.

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Wn**> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

<**lsb**> For the "32-bit" variant: is the bit number of the lsb of the source bitfield, in the range 0 to 31.

For the "64-bit" variant: is the bit number of the lsb of the source bitfield, in the range 0 to 63.

<**width**> For the "32-bit" variant: is the width of the bitfield, in the range 1 to 32-<lsb>.

For the "64-bit" variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Xn**> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Operational Information

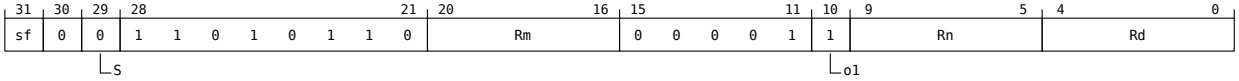
The description of SBFM gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.224 SDIV

Signed divide

This instruction divides the first signed source register value by the second signed source register value, and writes the result to the destination register. Dividing by zero writes the value zero to the destination register. The condition flags are not affected.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
SDIV <Wd>, <Wn>, <Wm>
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
SDIV <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
```

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wn**> Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Wm**> Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Xn**> Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Xm**> Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Operation

```
constant bits(datasize) operand1 = X[n, datasize];
constant bits(datasize) operand2 = X[m, datasize];
constant integer dividend = SInt(operand1);
constant integer divisor  = SInt(operand2);
integer result;
if divisor == 0 then
    result = 0;
elsif (dividend < 0) == (divisor < 0) then
    result = Abs(dividend) DIV Abs(divisor);    // same signs - positive result
else
    result = -(Abs(dividend) DIV Abs(divisor)); // different signs - negative result
X[d, datasize] = result<datasize-1:0>;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

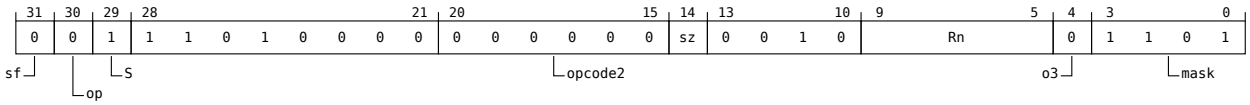
C6.2.225 SETF8, SETF16

Evaluation of 8-bit or 16-bit flag values

This instruction sets the PSTATE.NZV flags based on the value in the specified general-purpose register. `SETF8` treats the value as an 8-bit value. `SETF16` treats the value as a 16-bit value.

The PSTATE.C flag is not affected by these instructions.

Integer
(FEAT_FlagM)



Encoding for the SETF8 variant Applies when `(sz == 0)`

```
SETF8 <Wn>
```

Encoding for the SETF16 variant Applies when `(sz == 1)`

```
SETF16 <Wn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FlagM) then EndOfDecode(Decode_UNDEF);
constant integer size = 8 << UInt(sz);
constant integer n = UInt(Rn);
```

Assembler Symbols

<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

```
constant bits(32) reg = X[n, 32];
PSTATE.N = reg<size-1>;
PSTATE.Z = if (reg<size-1:0> == Zeros(size)) then '1' else '0';
PSTATE.V = reg<size> EOR reg<size-1>;
//PSTATE.C unchanged;
```

Operational information

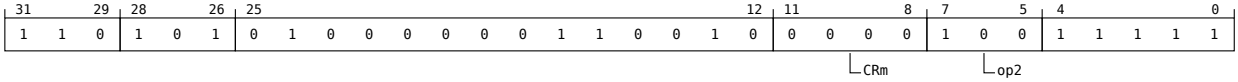
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.226 SEV

Send event

This instruction is a hint instruction that causes an event to be signaled to all PEs in the multiprocessor system. For more information, see Wait for Event mechanism and Send event.



Encoding

```
SEV
```

Decode for this encoding

```
// Empty.
```

Operation

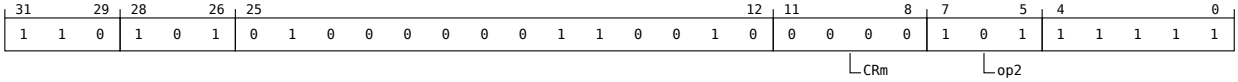
```
SendEvent();
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.227 SEVL

Send event local

This instruction is a hint instruction that causes an event to be signaled locally without requiring the event to be signaled to other PEs in the multiprocessor system. It can prime a wait-loop that starts with a `WFE` instruction.



Encoding

```
SEVL
```

Decode for this encoding

```
// Empty.
```

Operation

```
SendEventLocal();
```

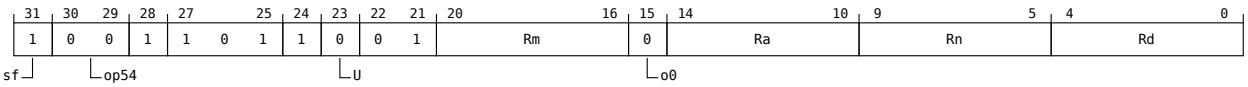

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.228 SMADDL

Signed multiply-add long

This instruction multiplies two 32-bit register values, adds a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias SMULL.



Encoding

```
SMADDL <Xd>, <Wn>, <Wm>, <Xa>
```

Decode for this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer a = UInt(Ra);
```

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

<Xa> Is the 64-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.

Alias Conditions

Alias	Is preferred when
SMULL	Ra == '11111'

Operation

```
constant bits(32) operand1 = X[n, 32];
constant bits(32) operand2 = X[m, 32];
constant bits(64) operand3 = X[a, 64];

constant integer result = SInt(operand3) + (SInt(operand1) * SInt(operand2));

X[d, 64] = result<63:0>;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.229 SMC

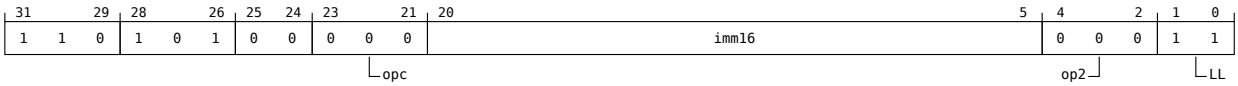
Secure monitor call

If execution of SMC is not trapped to EL2, this instruction is UNDEFINED.

SMC is available only for software executing at EL1. It is UNDEFINED at EL0.

If the value of HCR_EL2.TSC is 1, execution of an SMC instruction at EL1 generates a Secure Monitor Call exception, recording it in ESR_ELx, using the EC syndrome value 0x17, that is taken to EL2.

If the value of HCR_EL2.TSC is 0, the SMC instruction is UNDEFINED.



Encoding

```
SMC #<imm>
```

Decode for this encoding

```
constant bits(16) imm = imm16;
```

Assembler Symbols

<**imm**> Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the “imm16” field.

Operation

```
if PSTATE.EL == EL0 then UNDEFINED;
AArch64.CheckForSMCUndefOrTrap(imm);
Unreachable();
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

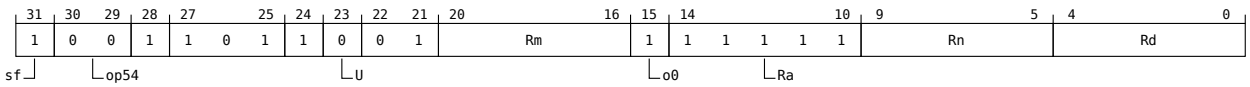
C6.2.230 SMNEGL

Signed multiply-negate long

This instruction multiplies two 32-bit register values, negates the product, and writes the result to the 64-bit destination register.

This is an alias of SMSUBL. This means:

- The encodings in this description are named to match the encodings of SMSUBL.
- The description of SMSUBL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding

SMNEGL <Xd>, <Wn>, <Wm>

is equivalent to

SMSUBL <Xd>, <Wn>, <Wm>, XZR

and is always the preferred disassembly.

Assembler Symbols

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Wn**> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<**Wm**> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

The description of SMSUBL gives the operational pseudocode for this instruction.

Operational Information

The description of SMSUBL gives the operational information for this instruction.

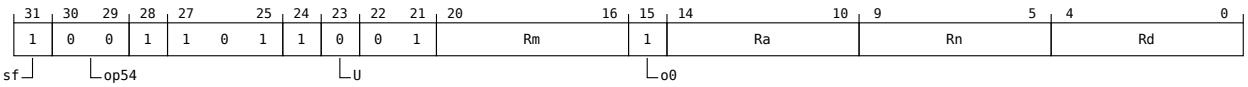
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.231 SMSUBL

Signed multiply-subtract long

This instruction multiplies two 32-bit register values, subtracts the product from a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias SMNEGL.



Encoding

```
SMSUBL <Xd>, <Wn>, <Wm>, <Xa>
```

Decode for this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer a = UInt(Ra);
```

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

<Xa> Is the 64-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.

Alias Conditions

Alias	Is preferred when
SMNEGL	`Ra == '11111'`

Operation

```
constant bits(32) operand1 = X[n, 32];
constant bits(32) operand2 = X[m, 32];
constant bits(64) operand3 = X[a, 64];

constant integer result = SInt(operand3) - (SInt(operand1) * SInt(operand2));

X[d, 64] = result<63:0>;
```

Operational information

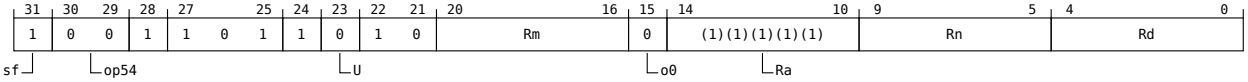
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.232 SMULH

Signed multiply high

This instruction multiplies two 64-bit register values, and writes bits[127:64] of the 128-bit result to the 64-bit destination register.



Encoding

```
SMULH <Xd>, <Xn>, <Xm>
```

Decode for this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
```

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

```
constant bits(64) operand1 = X[n, 64];
constant bits(64) operand2 = X[m, 64];

constant integer result = SInt(operand1) * SInt(operand2);

X[d, 64] = result<127:64>;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

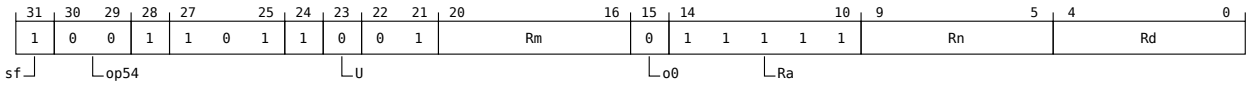
C6.2.233 SMULL

Signed multiply long

This instruction multiplies two 32-bit register values, and writes the result to the 64-bit destination register.

This is an alias of SMADDL. This means:

- The encodings in this description are named to match the encodings of SMADDL.
- The description of SMADDL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding

SMULL <Xd>, <Wn>, <Wm>

is equivalent to

SMADDL <Xd>, <Wn>, <Wm>, XZR

and is always the preferred disassembly.

Assembler Symbols

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wn**> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<**Wm**> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

Operation

The description of SMADDL gives the operational pseudocode for this instruction.

Operational Information

The description of SMADDL gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

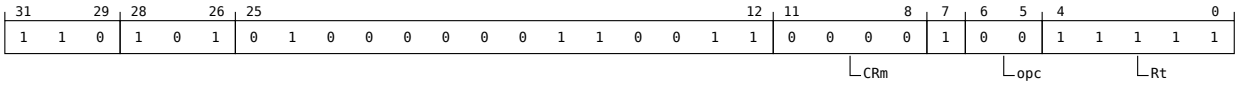
C6.2.234 SSBB

Speculative store bypass barrier

This instruction is a memory barrier that prevents speculative loads from bypassing earlier stores to the same virtual address under certain conditions. For more information and details of the semantics, see Speculative Store Bypass Barrier (SSBB).

This is an alias of DSB. This means:

- The encodings in this description are named to match the encodings of DSB.
- The description of DSB gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding

```
SSBB
```

is equivalent to

```
DSB #0
```

and is always the preferred disassembly.

Operation

The description of DSB gives the operational pseudocode for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.235 STCLR, STCLRL

Atomic bit clear on word or doubleword, without return

This instruction atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory.

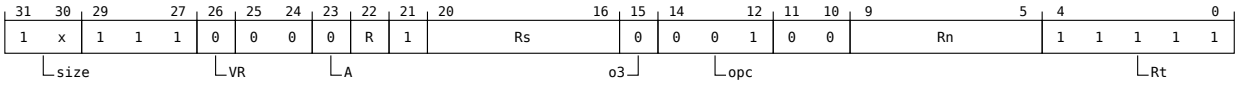
- `STCLR` does not have release semantics.
- `STCLRL` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDCLR, LDCLRA, LDCLRAL, LDCLRL. This means:

- The encodings in this description are named to match the encodings of LDCLR, LDCLRA, LDCLRAL, LDCLRL.
- The description of LDCLR, LDCLRA, LDCLRAL, LDCLRL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer
(FEAT_LSE)



Encoding for the 32-bit no memory ordering variant Applies when `(size == 10 && R == 0)`

```
STCLR <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDCLR <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 32-bit release variant Applies when `(size == 10 && R == 1)`

```
STCLRL <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDCLRL <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 64-bit no memory ordering variant Applies when `(size == 11 && R == 0)`

```
STCLR <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDCLR <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 64-bit release variant Applies when `(size == 11 && R == 1)`

```
STCLRL <Xs>, [<Xn|SP>]
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

is equivalent to

```
LDCLRL <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<**Ws**> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<**Xn**|**SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<**Xs**> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

Operation

The description of LDCLR, LDCLRA, LDCLRAL, LDCLRL gives the operational pseudocode for this instruction.

Operational Information

The description of LDCLR, LDCLRA, LDCLRAL, LDCLRL gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.236 STCLRB, STCLRLB

Atomic bit clear on byte, without return

This instruction atomically loads an 8-bit byte from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory.

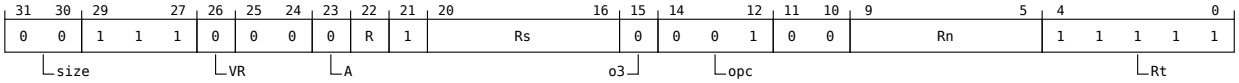
- `STCLRB` does not have release semantics.
- `STCLRLB` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB. This means:

- The encodings in this description are named to match the encodings of LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB.
- The description of LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer
(FEAT_LSE)



Encoding for the No memory ordering variant Applies when `(R == 0)`

```
STCLRB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDCLRB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the Release variant Applies when `(R == 1)`

```
STCLRLB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDCLRLB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<**Ws**> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB gives the operational pseudocode for this instruction.

Operational Information

The description of LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.237 STCLRH, STCLRLH

Atomic bit clear on halfword, without return

This instruction atomically loads a 16-bit halfword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory.

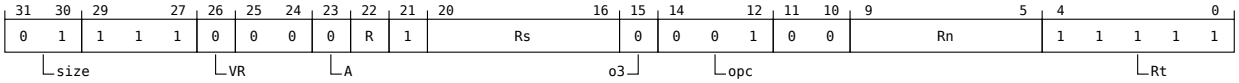
- `STCLRH` does not have release semantics.
- `STCLRLH` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH. This means:

- The encodings in this description are named to match the encodings of LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH.
- The description of LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer
(FEAT_LSE)



Encoding for the No memory ordering variant Applies when `(R == 0)`

```
STCLRH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDCLRH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the Release variant Applies when `(R == 1)`

```
STCLRLH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDCLRLH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<**Ws**> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH gives the operational pseudocode for this instruction.

Operational Information

The description of LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.238 STEOR, STEORL

Atomic exclusive-OR on word or doubleword, without return

This instruction atomically loads a 32-bit word or 64-bit doubleword from memory, performs an exclusive-OR with the value held in a register on it, and stores the result back to memory.

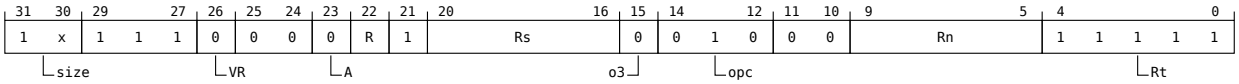
- `STEOR` does not have release semantics.
- `STEORL` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDEOR, LDEORA, LDEORAL, LDEORL. This means:

- The encodings in this description are named to match the encodings of LDEOR, LDEORA, LDEORAL, LDEORL.
- The description of LDEOR, LDEORA, LDEORAL, LDEORL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer
(FEAT_LSE)



Encoding for the 32-bit no memory ordering variant Applies when `(size == 10 && R == 0)`

```
STEOR <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDEOR <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 32-bit release variant Applies when `(size == 10 && R == 1)`

```
STEORL <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDEORL <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 64-bit no memory ordering variant Applies when `(size == 11 && R == 0)`

```
STEOR <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDEOR <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 64-bit release variant Applies when `(size == 11 && R == 1)`

```
STEORL <Xs>, [<Xn|SP>]
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

is equivalent to

```
LDEORL <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<**Ws**> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<**Xn**|**SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<**Xs**> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

Operation

The description of LDEOR, LDEORA, LDEORAL, LDEORL gives the operational pseudocode for this instruction.

Operational Information

The description of LDEOR, LDEORA, LDEORAL, LDEORL gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.239 STEORB, STEORLB

Atomic exclusive-OR on byte, without return

This instruction atomically loads an 8-bit byte from memory, performs an exclusive-OR with the value held in a register on it, and stores the result back to memory.

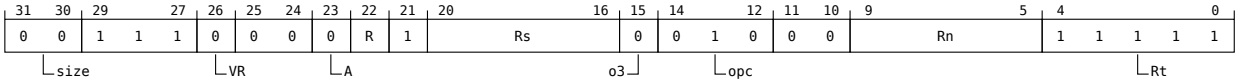
- STEORB does not have release semantics.
- STEORLB stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDEORB, LDEORAB, LDEORALB, LDEORLB. This means:

- The encodings in this description are named to match the encodings of LDEORB, LDEORAB, LDEORALB, LDEORLB.
- The description of LDEORB, LDEORAB, LDEORALB, LDEORLB gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer
(FEAT_LSE)



Encoding for the No memory ordering variant Applies when (R == 0)

```
STEORB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDEORB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the Release variant Applies when (R == 1)

```
STEORLB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDEORLB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<**Ws**> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDEORB, LDEORAB, LDEORALB, LDEORLB gives the operational pseudocode for this instruction.

Operational Information

The description of LDEORB, LDEORAB, LDEORALB, LDEORLB gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.240 STEORH, STEORLH

Atomic exclusive-OR on halfword, without return

This instruction atomically loads a 16-bit halfword from memory, performs an exclusive-OR with the value held in a register on it, and stores the result back to memory.

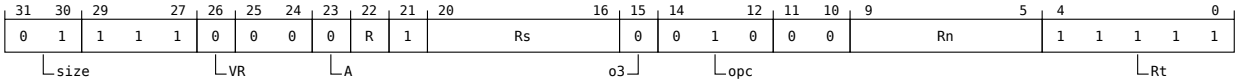
- `STEORH` does not have release semantics.
- `STEORLH` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDEORH, LDEORAH, LDEORALH, LDEORLH. This means:

- The encodings in this description are named to match the encodings of LDEORH, LDEORAH, LDEORALH, LDEORLH.
- The description of LDEORH, LDEORAH, LDEORALH, LDEORLH gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer
(FEAT_LSE)



Encoding for the No memory ordering variant Applies when `(R == 0)`

```
STEORH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDEORH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the Release variant Applies when `(R == 1)`

```
STEORLH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDEORLH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<**Ws**> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDEORH, LDEORAH, LDEORALH, LDEORLH gives the operational pseudocode for this instruction.

Operational Information

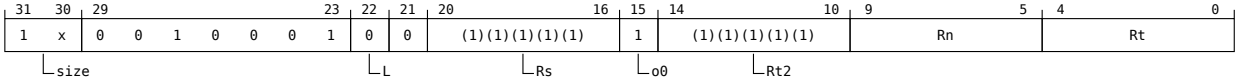
The description of LDEORH, LDEORAH, LDEORALH, LDEORLH gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.241 STLR

Store-release register

This instruction stores a 32-bit word or a 64-bit doubleword to a memory location, from a register. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about addressing modes, see Load/Store addressing modes.



Encoding for the 32-bit variant Applies when `(size == 10)`

```
STLR <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit variant Applies when `(size == 11)`

```
STLR <Xt>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant boolean wback = FALSE;
constant integer offset = 0;
constant boolean rt_unknown = FALSE;

constant integer elsize = 8 << UInt(size);
constant integer datasize = elsize;
constant boolean acquire = FALSE;
```

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

Operation

```
bits(64) address;
constant integer dbytes = datasize DIV 8;

constant AccessDescriptor accdesc = CreateAccDescAcqRel(MemOp_STORE, acquire, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);
bits(datasize) data;
if rt_unknown then
    data = bits(datasize) UNKNOWN;
else
    data = X[t, datasize];
Mem[address, dbytes, accdesc] = data;
if wback then
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if n == 31 then
    SP[64] = address;
else
    X[n, 64] = address;
```

Operational information

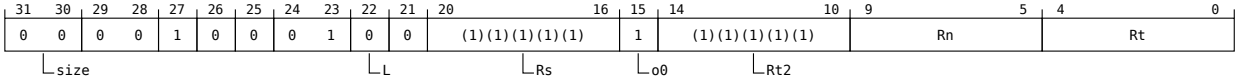
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.242 STLRB

Store-release register byte

This instruction stores a byte from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about addressing modes, see Load/Store addressing modes.



Encoding

```
STLRB <Wt>, [<Xn|SP>{, #0}]
```

Decode for this encoding

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant boolean acquire = FALSE;
```

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;

constant AccessDescriptor accdesc = CreateAccDescAcqRel(MemOp_STORE, acquire, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, 0, accdesc);
Mem[address, 1, accdesc] = X[t, 8];
```

Operational information

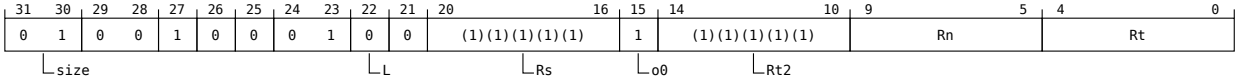
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.243 STLRH

Store-release register halfword

This instruction stores a halfword from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about addressing modes, see Load/Store addressing modes.



Encoding

```
STLRH <Wt>, [<Xn|SP>{, #0}]
```

Decode for this encoding

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant boolean acquire = FALSE;
```

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;

constant AccessDescriptor accdesc = CreateAccDescAcqRel(MemOp_STORE, acquire, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, 0, accdesc);
Mem[address, 2, accdesc] = X[t, 16];
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.244 STLUR

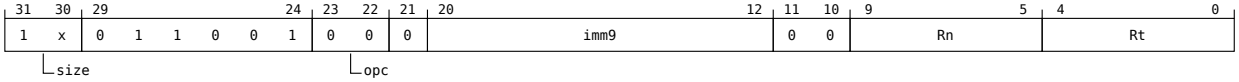
Store-release register (unscaled)

This instruction calculates an address from a base register value and an immediate offset, and stores a 32-bit word or a 64-bit doubleword to the calculated address, from a register.

The instruction has memory ordering semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release

For information about addressing modes, see Load/Store addressing modes.

Unscaled offset
(FEAT_LRCPC2)



Encoding for the 32-bit variant Applies when `(size == 10)`

```
STLUR <Wt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 64-bit variant Applies when `(size == 11)`

```
STLUR <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LRCPC2) then EndOfDecode(Decode_UNDEF);
constant integer scale = UInt(size);
constant bits(64) offset = SignExtend(imm9, 64);
constant integer n = UInt(Rn);
constant integer t = UInt(Rt);

constant integer datasize = 8 << scale;
constant boolean acquire = FALSE;
```

Assembler Symbols

<**Wt**> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<**simm**> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

<**Xt**> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

Operation

```
bits(64) address;

constant AccessDescriptor accdesc = CreateAccDescAcqRel(MemOp_STORE, acquire, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

Mem[address, datasize DIV 8, accdesc] = X[t, datasize];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.245 STLURB

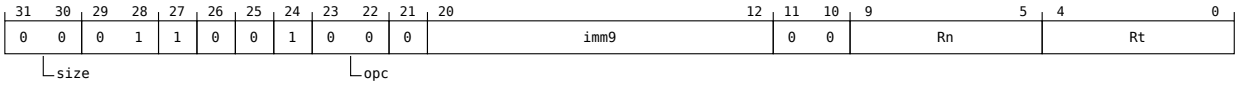
Store-release register byte (unscaled)

This instruction calculates an address from a base register value and an immediate offset, and stores a byte to the calculated address, from a 32-bit register.

The instruction has memory ordering semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release

For information about addressing modes, see Load/Store addressing modes.

Unscaled offset
(FEAT_LRCPC2)



Encoding

```
STLURB <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LRCPC2) then EndOfDecode(Decode_UNDEF);
constant bits(64) offset = SignExtend(imm9, 64);
constant integer n = UInt(Rn);
constant integer t = UInt(Rt);

constant integer datasize = 8;
constant boolean acquire = FALSE;
```

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

Operation

```
bits(64) address;

constant AccessDescriptor accdesc = CreateAccDescAcqRel(MemOp_STORE, acquire, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

Mem[address, datasize DIV 8, accdesc] = X[t, datasize];
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.246 STLURH

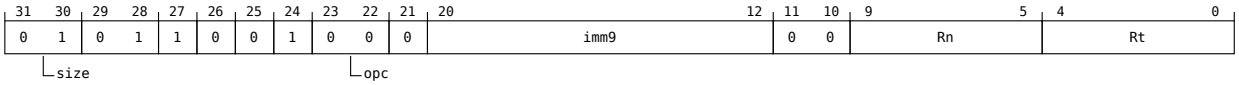
Store-release register halfword (unscaled)

This instruction calculates an address from a base register value and an immediate offset, and stores a halfword to the calculated address, from a 32-bit register.

The instruction has memory ordering semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release

For information about addressing modes, see Load/Store addressing modes.

Unscaled offset
(FEAT_LRCPC2)



Encoding

```
STLURH <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LRCPC2) then EndOfDecode(Decode_UNDEF);
constant bits(64) offset = SignExtend(imm9, 64);
constant integer n = UInt(Rn);
constant integer t = UInt(Rt);

constant integer datasize = 16;
constant boolean acquire = FALSE;
```

Assembler Symbols

<**Wt**> Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<**simm**> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

Operation

```
bits(64) address;

constant AccessDescriptor accdesc = CreateAccDescAcqRel(MemOp_STORE, acquire, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

Mem[address, datasize DIV 8, accdesc] = X[t, datasize];
```

Operational information

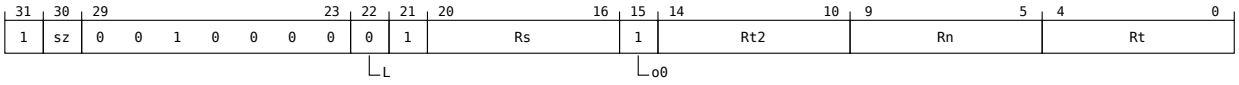
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.247 STLXP

Store-release exclusive pair of registers

This instruction stores two 32-bit words or two 64-bit doublewords to a memory location if the PE has exclusive access to the memory address, from two registers, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See Synchronization and semaphores. For information on single-copy atomicity and alignment requirements, see Requirements for single-copy atomicity and Alignment of data accesses. If a 64-bit pair Store-Exclusive succeeds, it causes a single-copy atomic update of the 128-bit memory location being updated. The instruction also has memory ordering semantics, as described in Load-Acquire, Store-Release. For information about addressing modes, see Load/Store addressing modes.



Encoding for the 32-bit variant Applies when `(sz == 0)`

```
STLXP <Ws>, <Wt1>, <Wt2>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit variant Applies when `(sz == 1)`

```
STLXP <Ws>, <Xt1>, <Xt2>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer t2 = UInt(Rt2);
constant integer n = UInt(Rn);

constant integer elsize = 32 << UInt(sz);
constant integer datasize = elsize * 2;
constant boolean acqrel = TRUE;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t || (s == t2) then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
if s == n && n != 31 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly STLXP.

Assembler Symbols

<**Ws**> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0 If the operation updates memory.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

1 If the operation fails to update memory.

<**Wt1**> Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<**Wt2**> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<**Xt1**> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<**Xt2**> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
bits(64) address;
bits(datasize) data;

constant integer dbytes = datasize DIV 8;
constant boolean privileged = PSTATE.EL != EL0;
constant boolean ispair = FALSE;
constant AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_STORE, acqrel, privileged, ispair, t,
                                                        t2);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n, 64];

if rt_unknown then
    data = bits(datasize) UNKNOWN;
else
    constant bits(datasize DIV 2) el1 = X[t, datasize DIV 2];
    constant bits(datasize DIV 2) el2 = X[t2, datasize DIV 2];
    data = if BigEndian(accdesc.acctype) then el1:el2 else el2:el1;
bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
// If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address,
// if accessed, would generate a synchronous Data Abort exception, it is
// IMPLEMENTATION DEFINED whether the exception is generated.
// It is a limitation of this model that synchronous Data Aborts are never
// generated in this case, as Mem[] is not called.

if AArch64.ExclusiveMonitorsPass(address, dbytes, accdesc) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    Mem[address, dbytes, accdesc] = data;
    status = ExclusiveMonitorsStatus();
X[s, 32] = ZeroExtend(status, 32);
```

Operational information

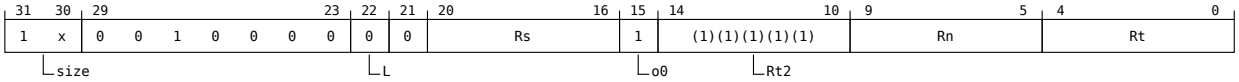
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.248 STLXR

Store-release exclusive register

This instruction stores a 32-bit word or a 64-bit doubleword to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See Synchronization and semaphores. The memory access is atomic. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about addressing modes, see Load/Store addressing modes.



Encoding for the 32-bit variant Applies when `(size == 10)`

```
STLXR <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit variant Applies when `(size == 11)`

```
STLXR <Ws>, <Xt>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer elsize = 8 << UInt(size);
constant boolean acqrel = TRUE;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
if s == n && n != 31 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly STLXR.

Assembler Symbols

<**Ws**> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the “Rs” field. The value returned is:

0 If the operation updates memory.

1 If the operation fails to update memory.

<**Wt**> Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<**Xt**> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
bits(64) address;
bits(elsize) data;

constant integer dbytes = elsize DIV 8;
constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_STORE, acqrel, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n, 64];

if rt_unknown then
    data = bits(elsize) UNKNOWN;
else
    data = X[t, elsize];
bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
// If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address,
// if accessed, would generate a synchronous Data Abort exception, it is
// IMPLEMENTATION DEFINED whether the exception is generated.
// It is a limitation of this model that synchronous Data Aborts are never
// generated in this case, as Mem[] is not called.

if AArch64.ExclusiveMonitorsPass(address, dbytes, accdesc) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, dbytes, accdesc] = data;
    status = ExclusiveMonitorsStatus();
X[s, 32] = ZeroExtend(status, 32);
```

Operational information

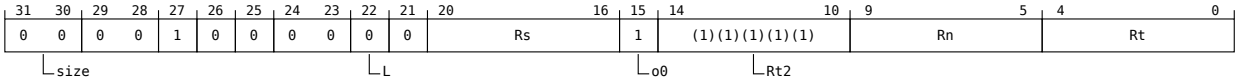
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.249 STLXRB

Store-release exclusive register byte

This instruction stores a byte from a 32-bit register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See Synchronization and semaphores. The memory access is atomic. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about addressing modes, see Load/Store addressing modes.



Encoding

```
STLXRB <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Decode for this encoding

```
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acqrel = TRUE;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
if s == n && n != 31 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly STLXRB.

Assembler Symbols

<**Ws**> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0 If the operation updates memory.

1 If the operation fails to update memory.

<**Wt**> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts

If a synchronous Data Abort exception is generated by the execution of this instruction:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Memory is not updated.
- <Ws> is not updated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
bits(64) address;
bits(8) data;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_STORE, acqrel, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n, 64];

if rt_unknown then
    data = bits(8) UNKNOWN;
else
    data = X[t, 8];
bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
// If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address,
// if accessed, would generate a synchronous Data Abort exception, it is
// IMPLEMENTATION DEFINED whether the exception is generated.
// It is a limitation of this model that synchronous Data Aborts are never
// generated in this case, as Mem[] is not called.

if AArch64.ExclusiveMonitorsPass(address, 1, accdesc) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, 1, accdesc] = data;
    status = ExclusiveMonitorsStatus();
X[s, 32] = ZeroExtend(status, 32);
```

Operational information

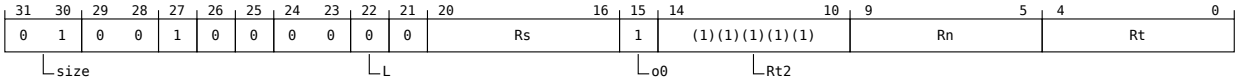
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.250 STLXRH

Store-release exclusive register halfword

This instruction stores a halfword from a 32-bit register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See Synchronization and semaphores. The memory access is atomic. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about addressing modes, see Load/Store addressing modes.



Encoding

```
STLXRH <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Decode for this encoding

```
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acqrel = TRUE;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
if s == n && n != 31 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly STLXRH.

Assembler Symbols

<**Ws**> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0 If the operation updates memory.

1 If the operation fails to update memory.

<**Wt**> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Memory is not updated.
- <Ws> is not updated.

A non halfword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
bits(64) address;
bits(16) data;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_STORE, acqrel, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n, 64];

if rt_unknown then
    data = bits(16) UNKNOWN;
else
    data = X[t, 16];
bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
// If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address,
// if accessed, would generate a synchronous Data Abort exception, it is
// IMPLEMENTATION DEFINED whether the exception is generated.
// It is a limitation of this model that synchronous Data Aborts are never
// generated in this case, as Mem[] is not called.

if AArch64.ExclusiveMonitorsPass(address, 2, accdesc) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, 2, accdesc] = data;
    status = ExclusiveMonitorsStatus();
X[s, 32] = ZeroExtend(status, 32);
```

Operational information

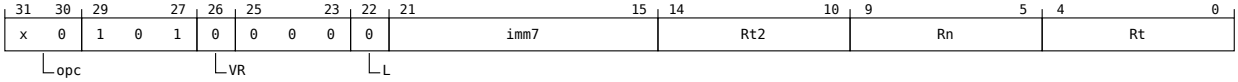
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.251 STNP

Store pair of registers, with non-temporal hint

This instruction calculates an address from a base register value and an immediate offset, and stores two 32-bit words or two 64-bit doublewords to the calculated address, from two registers. For information about addressing modes, see Load/Store addressing modes. For information about non-temporal pair instructions, see Load/Store non-temporal pair.



Encoding for the 32-bit variant Applies when `(opc == 00)`

```
STNP <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]
```

Encoding for the 64-bit variant Applies when `(opc == 10)`

```
STNP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

Decode for all variants of this encoding

```
constant integer t = UInt(Rt);
constant integer t2 = UInt(Rt2);
constant integer n = UInt(Rn);
constant boolean nontemporal = TRUE;
constant integer scale = 2 + UInt(opc<1>);
constant integer datasize = 8 << scale;
constant bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```

Assembler Symbols

<Wt1> Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Wt2> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> For the "32-bit" variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.

For the "64-bit" variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

<Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

Operation

```
bits(64) address;
constant integer dbytes = datasize DIV 8;
constant boolean privileged = PSTATE.EL != EL0;
constant boolean ispair = TRUE;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged, ispair,
                                                     t, t2);
if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);
constant bits(2*datasize) data = (if BigEndian(accdesc.acctype)
                                  then X[t,  datasize]:X[t2, datasize]
                                  else X[t2, datasize]:X[t,  datasize]);
Mem[address, 2 * dbytes, accdesc] = data;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

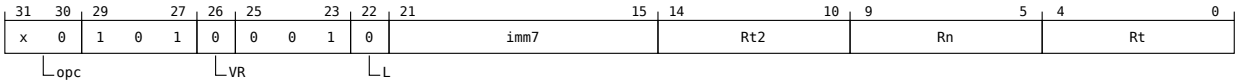
C6.2.252 STP

Store pair of registers

This instruction calculates an address from a base register value and an immediate offset, and stores two 32-bit words or two 64-bit doublewords to the calculated address, from two registers. For information about addressing modes, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index , Pre-index and Signed offset

Post-index



Encoding for the 32-bit variant Applies when `(opc == 00)`

```
STP <Wt1>, <Wt2>, [<Xn|SP>], #<imm>
```

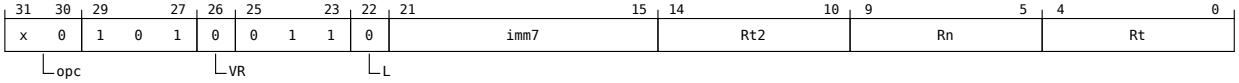
Encoding for the 64-bit variant Applies when `(opc == 10)`

```
STP <Xt1>, <Xt2>, [<Xn|SP>], #<imm>
```

Decode for all variants of this encoding

```
constant boolean wback = TRUE;
constant boolean postindex = TRUE;
```

Pre-index



Encoding for the 32-bit variant Applies when `(opc == 00)`

```
STP <Wt1>, <Wt2>, [<Xn|SP>, #<imm>]!
```

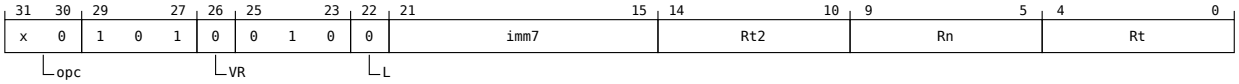
Encoding for the 64-bit variant Applies when `(opc == 10)`

```
STP <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]!
```

Decode for all variants of this encoding

```
constant boolean wback = TRUE;
constant boolean postindex = FALSE;
```

Signed offset



Encoding for the 32-bit variant Applies when `(opc == 00)`

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
STP <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]
```

Encoding for the 64-bit variant Applies when `(opc == 10)`

```
STP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

Decode for all variants of this encoding

```
constant boolean wback = FALSE;
constant boolean postindex = FALSE;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly STP.

Assembler Symbols

<**Wt1**> Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<**Wt2**> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<**imm**> For the "Post-index 32-bit" and "Pre-index 32-bit" variants: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4.

For the "Post-index 64-bit" and "Pre-index 64-bit" variants: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as <imm>/8.

For the "Signed offset 32-bit" variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.

For the "Signed offset 64-bit" variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

<**Xt1**> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<**Xt2**> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer t2 = UInt(Rt2);
constant integer n = UInt(Rn);
constant boolean nontemporal = FALSE;
constant integer scale = 2 + UInt(opc<1>);
constant integer datasize = 8 << scale;
constant bits(64) offset = LSL(SignExtend(imm7, 64), scale);

boolean rt_unknown = FALSE;
if wback && (t == n || t2 == n) && n != 31 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;   // Value stored is pre-writeback
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // Value stored is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
```

Operation

```
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant boolean privileged = PSTATE.EL != EL0;
constant boolean ispair = TRUE;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged, ispair,
                                                     t, t2);
if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

if !postindex then
    address = AddressAdd(address, offset, accdesc);

if rt_unknown && t == n then
    data1 = bits(datasize) UNKNOWN;
else
    data1 = X[t, datasize];
if rt_unknown && t2 == n then
    data2 = bits(datasize) UNKNOWN;
else
    data2 = X[t2, datasize];

constant bits(2*datasize) data = (if BigEndian(accdesc.acctype) then data1:data2
                                                                else data2:data1);
Mem[address, 2 * dbytes, accdesc] = data;

if wback then
    if postindex then
        address = AddressAdd(address, offset, accdesc);
    if n == 31 then
        SP[64] = address;
    else
        X[n, 64] = address;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

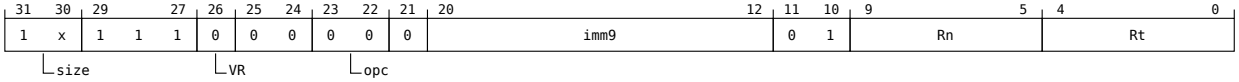
C6.2.253 STR (immediate)

Store register (immediate)

This instruction stores a word or a doubleword from a register to memory. The address that is used for the store is calculated from a base register and an immediate offset. For information about addressing modes, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index , Pre-index and Unsigned offset

Post-index



Encoding for the 32-bit variant Applies when `(size == 10)`

```
STR <Wt>, [<Xn|SP>], #<simm>
```

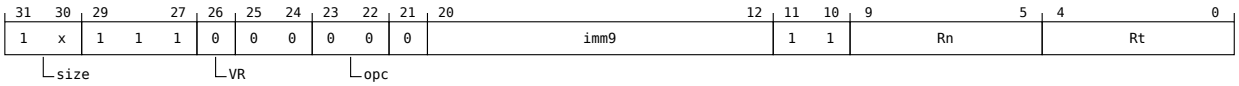
Encoding for the 64-bit variant Applies when `(size == 11)`

```
STR <Xt>, [<Xn|SP>], #<simm>
```

Decode for all variants of this encoding

```
constant boolean wback = TRUE;
constant boolean postindex = TRUE;
constant integer scale = UInt(size);
constant bits(64) offset = SignExtend(imm9, 64);
```

Pre-index



Encoding for the 32-bit variant Applies when `(size == 10)`

```
STR <Wt>, [<Xn|SP>, #<simm>]!
```

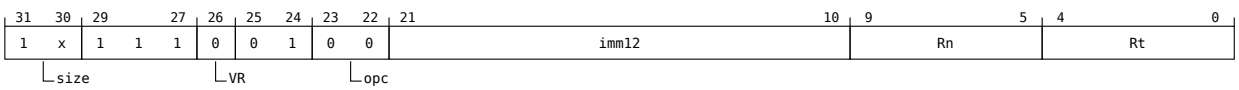
Encoding for the 64-bit variant Applies when `(size == 11)`

```
STR <Xt>, [<Xn|SP>, #<simm>]!
```

Decode for all variants of this encoding

```
constant boolean wback = TRUE;
constant boolean postindex = FALSE;
constant integer scale = UInt(size);
constant bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset



Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 32-bit variant Applies when `(size == 10)`

```
STR <Wt>, [<Xn|SP>{, #<pimm>}]
```

Encoding for the 64-bit variant Applies when `(size == 11)`

```
STR <Xt>, [<Xn|SP>{, #<pimm>}]
```

Decode for all variants of this encoding

```
constant boolean wback = FALSE;
constant boolean postindex = FALSE;
constant integer scale = UInt(size);
constant bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

Assembler Symbols

<**Wt**> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<**simm**> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.

<**Xt**> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<**pimm**> For the "32-bit" variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4.

For the "64-bit" variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer datasize = 8 << scale;
constant boolean nontemporal = FALSE;

Constraint c;
boolean rt_unknown = FALSE;
if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE    rt_unknown = FALSE;    // Value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE;     // Value stored is UNKNOWN
        when Constraint_UNDEF   EndOfDecode(Decode_UNDEF);
        when Constraint_NOP     EndOfDecode(Decode_NOP);
```

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

if !postindex then
    address = AddressAdd(address, offset, accdesc);

bits(datasize) data;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if rt_unknown then
    data = bits(datasize) UNKNOWN;
else
    data = X[t, datasize];

Mem[address, datasize DIV 8, accdesc] = data;

if wback then
    if postindex then
        address = AddressAdd(address, offset, accdesc);
    if n == 31 then
        SP[64] = address;
    else
        X[n, 64] = address;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

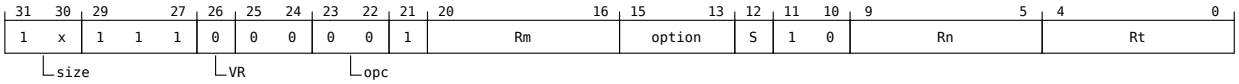
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.254 STR (register)

Store register (register)

This instruction calculates an address from a base register value and an offset register value, and stores a 32-bit word or a 64-bit doubleword to the calculated address, from a register. For information about addressing modes, see Load/Store addressing modes.

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.



Encoding for the 32-bit variant Applies when `(size == 10)`

```
STR <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Encoding for the 64-bit variant Applies when `(size == 11)`

```
STR <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Decode for all variants of this encoding

```
if option<1> == '0' then EndOfDecode(Decode_UNDEF);             // sub-word index
constant ExtendType extend_type = DecodeRegExtend(option);
constant integer scale = UInt(size);
constant integer shift = if S == '1' then scale else 0;
```

Assembler Symbols

<**Wt**> Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<**Wm**> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the “Rm” field.

<**Xm**> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the “Rm” field.

<**extend**> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted, encoded in “option”:

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<**amount**> For the “32-bit” variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

S	<amount>
0	#0
1	#2

For the “64-bit” variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#3

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 8 << scale;
constant boolean nontemporal = FALSE;
```

Operation

```
constant bits(64) offset = ExtendReg(m, extend_type, shift, 64);
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

Mem[address, datasize DIV 8, accdesc] = X[t, datasize];
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

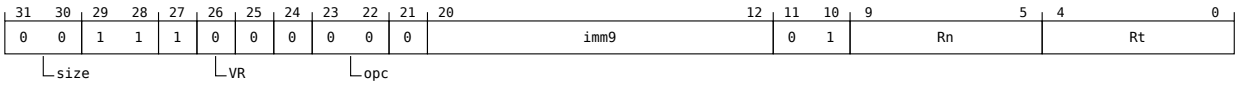
C6.2.255 STRB (immediate)

Store register byte (immediate)

This instruction stores the least significant byte of a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset. For information about addressing modes, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index , Pre-index and Unsigned offset

Post-index



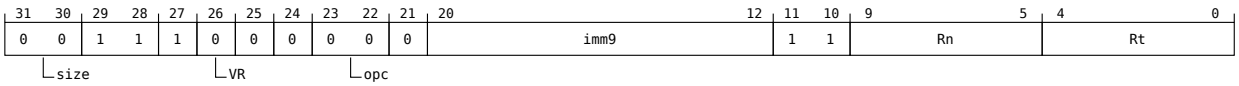
Encoding

```
STRB <Wt>, [<Xn|SP>], #<simm>
```

Decode for this encoding

```
constant boolean wback = TRUE;
constant boolean postindex = TRUE;
constant bits(64) offset = SignExtend(imm9, 64);
```

Pre-index



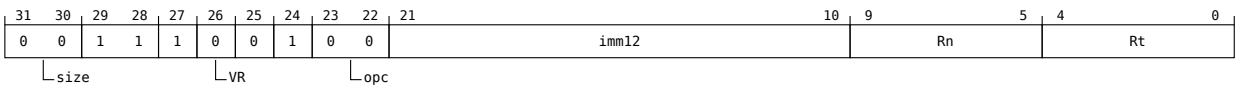
Encoding

```
STRB <Wt>, [<Xn|SP>, #<simm>]!
```

Decode for this encoding

```
constant boolean wback = TRUE;
constant boolean postindex = FALSE;
constant bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset



Encoding

```
STRB <Wt>, [<Xn|SP>{, #<pimm>}]
```

Decode for this encoding

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant boolean wback = FALSE;
constant boolean postindex = FALSE;
constant bits(64) offset = LSL(ZeroExtend(imm12, 64), 0);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly STRB (immediate).

Assembler Symbols

<**Wt**> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<**simm**> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.

<**pimm**> Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant boolean nontemporal = FALSE;

Constraint c;
boolean rt_unknown = FALSE;
if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE    rt_unknown = FALSE;    // Value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE;     // Value stored is UNKNOWN
        when Constraint_UNDEF   EndOfDecode(Decode_UNDEF);
        when Constraint_NOP     EndOfDecode(Decode_NOP);
```

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

if !postindex then
    address = AddressAdd(address, offset, accdesc);

bits(8) data;
if rt_unknown then
    data = bits(8) UNKNOWN;
else
    data = X[t, 8];

Mem[address, 1, accdesc] = data;

if wback then
    if postindex then
        address = AddressAdd(address, offset, accdesc);
    if n == 31 then
        SP[64] = address;
    else
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        X[n, 64] = address;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

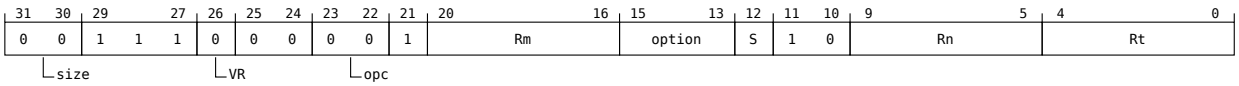
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.256 STRB (register)

Store register byte (register)

This instruction calculates an address from a base register value and an offset register value, and stores a byte from a 32-bit register to the calculated address. For information about addressing modes, see Load/Store addressing modes.

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.



Encoding for the Extended register variant Applies when `(option != 011)`

```
STRB <Wt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

Encoding for the Shifted register variant Applies when `(option == 011)`

```
STRB <Wt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

Decode for all variants of this encoding

```
if option<1> == '0' then EndOfDecode(Decode_UNDEF);             // sub-word index
constant ExtendType extend_type = DecodeRegExtend(option);
constant integer shift = 0;
```

Assembler Symbols

<**Wt**> Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<**Wm**> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the “Rm” field.

<**Xm**> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the “Rm” field.

<**extend**> Is the index extend specifier, encoded in “option”:

option	<extend>
010	UXTW
110	SXTW
111	SXTX

<**amount**> Is the index shift amount, it must be #0, encoded in “S” as 0 if omitted, or as 1 if present.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant boolean nontemporal = FALSE;
```

Operation

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant bits(64) offset = ExtendReg(m, extend_type, shift, 64);
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

Mem[address, 1, accdesc] = X[t, 8];
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

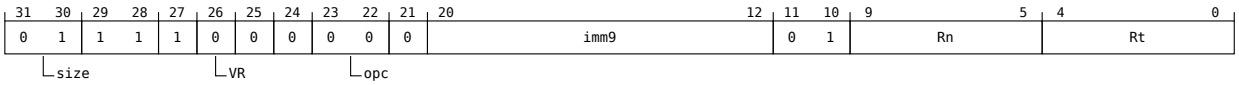
C6.2.257 STRH (immediate)

Store register halfword (immediate)

This instruction stores the least significant halfword of a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset. For information about addressing modes, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index , Pre-index and Unsigned offset

Post-index



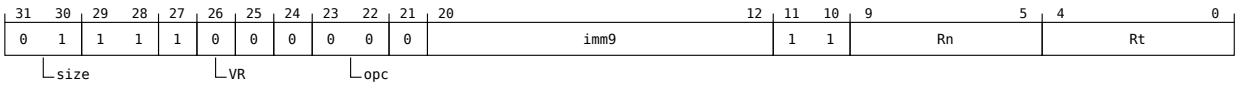
Encoding

```
STRH <Wt>, [<Xn|SP>], #<simm>
```

Decode for this encoding

```
constant boolean wback = TRUE;
constant boolean postindex = TRUE;
constant bits(64) offset = SignExtend(imm9, 64);
```

Pre-index



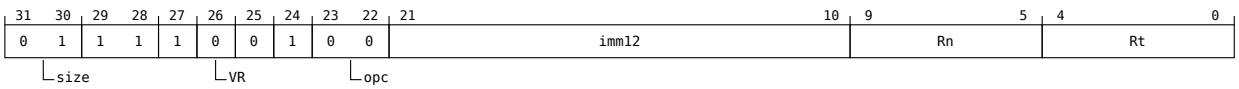
Encoding

```
STRH <Wt>, [<Xn|SP>, #<simm>]!
```

Decode for this encoding

```
constant boolean wback = TRUE;
constant boolean postindex = FALSE;
constant bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset



Encoding

```
STRH <Wt>, [<Xn|SP>{, #<pimm>}]
```

Decode for this encoding

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant boolean wback = FALSE;
constant boolean postindex = FALSE;
constant bits(64) offset = LSL(ZeroExtend(imm12, 64), 1);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly STRH (immediate).

Assembler Symbols

<**Wt**> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<**simm**> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.

<**pimm**> Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant boolean nontemporal = FALSE;

Constraint c;
boolean rt_unknown = FALSE;
if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE    rt_unknown = FALSE;    // Value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE;     // Value stored is UNKNOWN
        when Constraint_UNDEF   EndOfDecode(Decode_UNDEF);
        when Constraint_NOP     EndOfDecode(Decode_NOP);
```

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

if !postindex then
    address = AddressAdd(address, offset, accdesc);

bits(16) data;
if rt_unknown then
    data = bits(16) UNKNOWN;
else
    data = X[t, 16];

Mem[address, 2, accdesc] = data;

if wback then
    if postindex then
        address = AddressAdd(address, offset, accdesc);
    if n == 31 then
        SP[64] = address;
    else
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        X[n, 64] = address;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

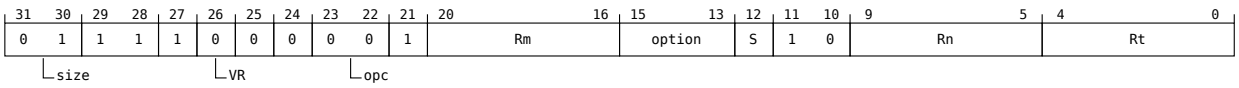
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.258 STRH (register)

Store register halfword (register)

This instruction calculates an address from a base register value and an offset register value, and stores a halfword from a 32-bit register to the calculated address. For information about addressing modes, see Load/Store addressing modes.

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.



Encoding

```
STRH <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Decode for this encoding

```
if option<1> == '0' then EndOfDecode(Decode_UNDEF);             // sub-word index
constant ExtendType extend_type = DecodeRegExtend(option);
constant integer shift = if S == '1' then 1 else 0;
```

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted, encoded in "option":

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<amount> Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

S	<amount>
0	#0
1	#1

Shared Decode

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant boolean nontemporal = FALSE;
```

Operation

```
constant bits(64) offset = ExtendReg(m, extend_type, shift, 64);
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

Mem[address, 2, accdesc] = X[t, 16];
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.259 STSET, STSETL

Atomic bit set on word or doubleword, without return

This instruction atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory.

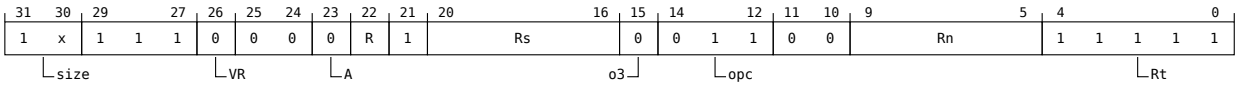
- `STSET` does not have release semantics.
- `STSETL` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDSET, LDSETA, LDSETAL, LDSETL. This means:

- The encodings in this description are named to match the encodings of LDSET, LDSETA, LDSETAL, LDSETL.
- The description of LDSET, LDSETA, LDSETAL, LDSETL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer
(FEAT_LSE)



Encoding for the 32-bit no memory ordering variant Applies when `(size == 10 && R == 0)`

```
STSET <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSET <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 32-bit release variant Applies when `(size == 10 && R == 1)`

```
STSETL <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSETL <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 64-bit no memory ordering variant Applies when `(size == 11 && R == 0)`

```
STSET <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDSET <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 64-bit release variant Applies when `(size == 11 && R == 1)`

```
STSETL <Xs>, [<Xn|SP>]
```

is equivalent to

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
LDSETL <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<**Ws**> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<**Xs**> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

Operation

The description of LDSET, LDSETA, LDSETAL, LDSETL gives the operational pseudocode for this instruction.

Operational Information

The description of LDSET, LDSETA, LDSETAL, LDSETL gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.260 STSETB, STSETLB

Atomic bit set on byte, without return

This instruction atomically loads an 8-bit byte from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory.

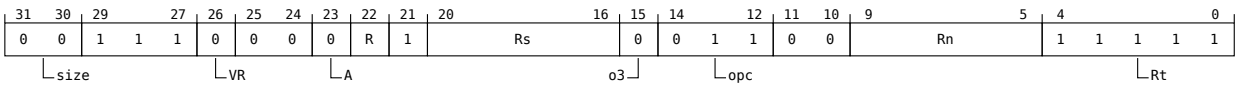
- `STSETB` does not have release semantics.
- `STSETLB` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDSETB, LDSETAB, LDSETALB, LDSETLB. This means:

- The encodings in this description are named to match the encodings of LDSETB, LDSETAB, LDSETALB, LDSETLB.
- The description of LDSETB, LDSETAB, LDSETALB, LDSETLB gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer
(FEAT_LSE)



Encoding for the No memory ordering variant Applies when `(R == 0)`

```
STSETB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSETB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the Release variant Applies when `(R == 1)`

```
STSETLB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSETLB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<**Ws**> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSETB, LDSETAB, LDSETALB, LDSETLB gives the operational pseudocode for this instruction.

Operational Information

The description of LDSETB, LDSETAB, LDSETALB, LDSETLB gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.261 STSETH, STSETLH

Atomic bit set on halfword, without return

This instruction atomically loads a 16-bit halfword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory.

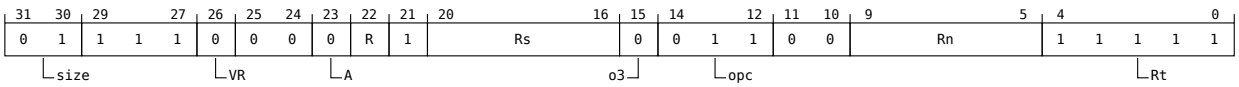
- `STSETH` does not have release semantics.
- `STSETLH` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDSETH, LDSETAH, LDSETALH, LDSETLH. This means:

- The encodings in this description are named to match the encodings of LDSETH, LDSETAH, LDSETALH, LDSETLH.
- The description of LDSETH, LDSETAH, LDSETALH, LDSETLH gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer
(FEAT_LSE)



Encoding for the No memory ordering variant Applies when `(R == 0)`

```
STSETH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSETH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the Release variant Applies when `(R == 1)`

```
STSETLH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSETLH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<**Ws**> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSETH, LDSETAH, LDSETALH, LDSETLH gives the operational pseudocode for this instruction.

Operational Information

The description of LDSETH, LDSETAH, LDSETALH, LDSETLH gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.262 STSMAX, STSMAXL

Atomic signed maximum on word or doubleword, without return

This instruction atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers.

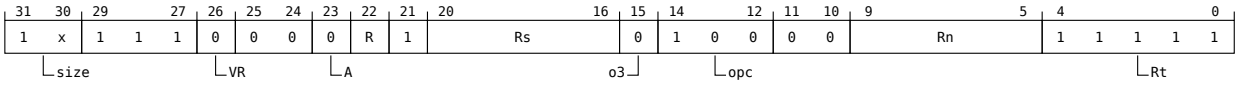
- STSMAX does not have release semantics.
- STSMAXL stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL. This means:

- The encodings in this description are named to match the encodings of LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL.
- The description of LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer
(FEAT_LSE)



Encoding for the 32-bit no memory ordering variant Applies when `(size == 10 && R == 0)`

```
STSMAX <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMAX <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 32-bit release variant Applies when `(size == 10 && R == 1)`

```
STSMAXL <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMAXL <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 64-bit no memory ordering variant Applies when `(size == 11 && R == 0)`

```
STSMAX <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDSMAX <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 64-bit release variant Applies when `(size == 11 && R == 1)`

```
STSMAXL <Xs>, [<Xn|SP>]
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

is equivalent to

```
LDSMAXL <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<**Ws**> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<**Xn**|**SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<**Xs**> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

Operation

The description of LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL gives the operational pseudocode for this instruction.

Operational Information

The description of LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.263 STSMAXB, STSMAXLB

Atomic signed maximum on byte, without return

This instruction atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers.

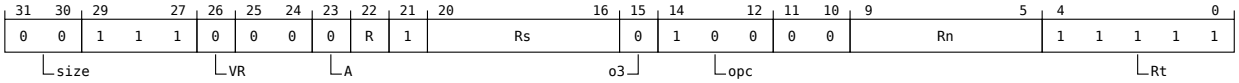
- `STSMAXB` does not have release semantics.
- `STSMAXLB` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB. This means:

- The encodings in this description are named to match the encodings of LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB.
- The description of LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer
(FEAT_LSE)



Encoding for the No memory ordering variant Applies when `(R == 0)`

```
STSMAXB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMAXB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the Release variant Applies when `(R == 1)`

```
STSMAXLB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMAXLB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<**Ws**> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB gives the operational pseudocode for this instruction.

Operational Information

The description of LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.264 STSMAXH, STSMAXLH

Atomic signed maximum on halfword, without return

This instruction atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers.

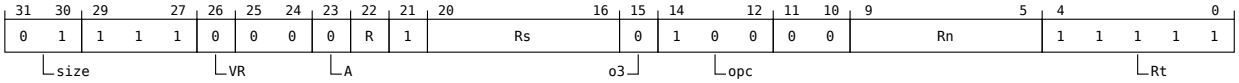
- `STSMAXH` does not have release semantics.
- `STSMAXLH` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH. This means:

- The encodings in this description are named to match the encodings of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH.
- The description of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer
(FEAT_LSE)



Encoding for the No memory ordering variant Applies when `(R == 0)`

```
STSMAXH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMAXH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the Release variant Applies when `(R == 1)`

```
STSMAXLH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMAXLH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<**Ws**> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH gives the operational pseudocode for this instruction.

Operational Information

The description of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.265 STSMIN, STSMINL

Atomic signed minimum on word or doubleword, without return

This instruction atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers.

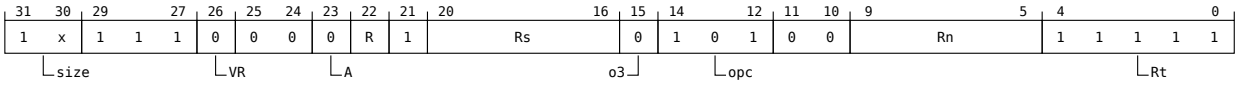
- `STSMIN` does not have release semantics.
- `STSMINL` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDSMIN, LDSMINA, LDSMINAL, LDSMINL. This means:

- The encodings in this description are named to match the encodings of LDSMIN, LDSMINA, LDSMINAL, LDSMINL.
- The description of LDSMIN, LDSMINA, LDSMINAL, LDSMINL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer
(FEAT_LSE)



Encoding for the 32-bit no memory ordering variant Applies when `(size == 10 && R == 0)`

```
STSMIN <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMIN <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 32-bit release variant Applies when `(size == 10 && R == 1)`

```
STSMINL <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMINL <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 64-bit no memory ordering variant Applies when `(size == 11 && R == 0)`

```
STSMIN <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDSMIN <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 64-bit release variant Applies when `(size == 11 && R == 1)`

```
STSMINL <Xs>, [<Xn|SP>]
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

is equivalent to

```
LDSMINL <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<**Ws**> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<**Xn**|**SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<**Xs**> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

Operation

The description of LDSMIN, LDSMINA, LDSMINAL, LDSMINL gives the operational pseudocode for this instruction.

Operational Information

The description of LDSMIN, LDSMINA, LDSMINAL, LDSMINL gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.266 STSMINB, STSMINLB

Atomic signed minimum on byte, without return

This instruction atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers.

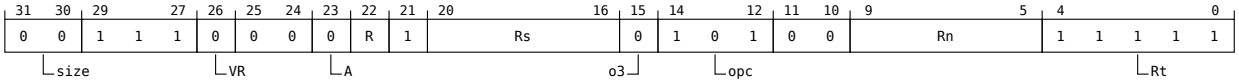
- `STSMINB` does not have release semantics.
- `STSMINLB` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB. This means:

- The encodings in this description are named to match the encodings of LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB.
- The description of LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer
(FEAT_LSE)



Encoding for the No memory ordering variant Applies when `(R == 0)`

```
STSMINB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMINB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the Release variant Applies when `(R == 1)`

```
STSMINLB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMINLB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<**Ws**> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB gives the operational pseudocode for this instruction.

Operational Information

The description of LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.267 STSMINH, STSMINLH

Atomic signed minimum on halfword, without return

This instruction atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers.

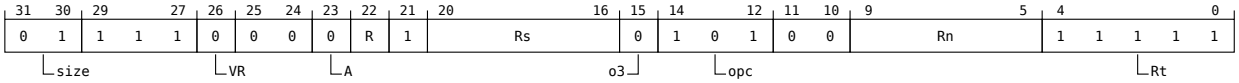
- `STSMINH` does not have release semantics.
- `STSMINLH` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH. This means:

- The encodings in this description are named to match the encodings of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH.
- The description of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer
(FEAT_LSE)



Encoding for the No memory ordering variant Applies when `(R == 0)`

```
STSMINH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMINH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the Release variant Applies when `(R == 1)`

```
STSMINLH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMINLH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<**Ws**> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH gives the operational pseudocode for this instruction.

Operational Information

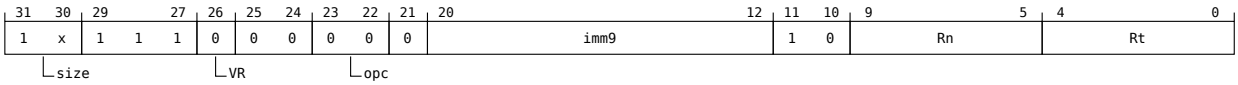
The description of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.268 STTR

Store register (unprivileged)

This instruction stores a word or doubleword from a register to memory. The address that is used for the store is calculated from a base register and an immediate offset.



Encoding for the 32-bit variant Applies when `(size == 10)`

```
STTR <Wt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 64-bit variant Applies when `(size == 11)`

```
STTR <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
constant integer scale = UInt(size);
constant bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer datasize = 8 << scale;
constant boolean nontemporal = FALSE;
```

Operation

```
bits(64) address;

constant boolean privileged = AArch64.IsUnprivAccessPriv();
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

Mem[address, datasize DIV 8, accdesc] = X[t, datasize];
```

Operational information

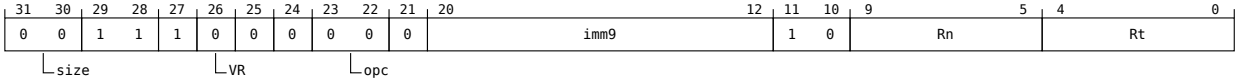
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.269 STTRB

Store register byte (unprivileged)

This instruction stores a byte from a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset.



Encoding

```
STTRB <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
constant bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<**Wt**> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<**simm**> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer datasize = 8;
constant boolean nontemporal = FALSE;
```

Operation

```
bits(64) address;

constant boolean privileged = AArch64.IsUnprivAccessPriv();
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

Mem[address, datasize DIV 8, accdesc] = X[t, datasize];
```

Operational information

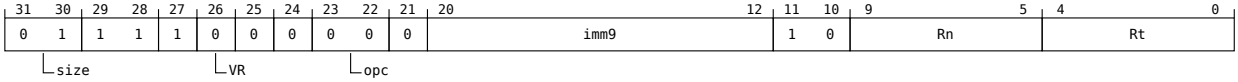
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.270 STTRH

Store register halfword (unprivileged)

This instruction stores a halfword from a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset.



Encoding

```
STTRH <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
constant bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<**Wt**> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<**simm**> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer datasize = 16;
constant boolean nontemporal = FALSE;
```

Operation

```
bits(64) address;

constant boolean privileged = AArch64.IsUnprivAccessPriv();
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

Mem[address, datasize DIV 8, accdesc] = X[t, datasize];
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.271 STUMAX, STUMAXL

Atomic unsigned maximum on word or doubleword, without return

This instruction atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers.

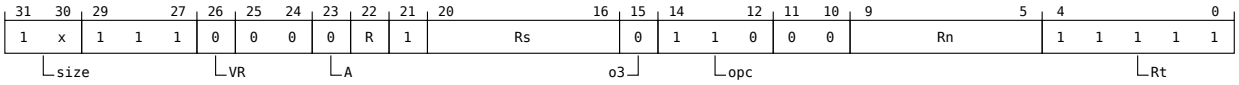
- STUMAX does not have release semantics.
- STUMAXL stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL. This means:

- The encodings in this description are named to match the encodings of LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL.
- The description of LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer
(FEAT_LSE)



Encoding for the 32-bit no memory ordering variant Applies when `(size == 10 && R == 0)`

```
STUMAX <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMAX <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 32-bit release variant Applies when `(size == 10 && R == 1)`

```
STUMAXL <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMAXL <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 64-bit no memory ordering variant Applies when `(size == 11 && R == 0)`

```
STUMAX <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDUMAX <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 64-bit release variant Applies when `(size == 11 && R == 1)`

```
STUMAXL <Xs>, [<Xn|SP>]
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

is equivalent to

```
LDUMAXL <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<**Ws**> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<**Xn**|**SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<**Xs**> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

Operation

The description of LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL gives the operational pseudocode for this instruction.

Operational Information

The description of LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.272 STUMAXB, STUMAXLB

Atomic unsigned maximum on byte, without return

This instruction atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers.

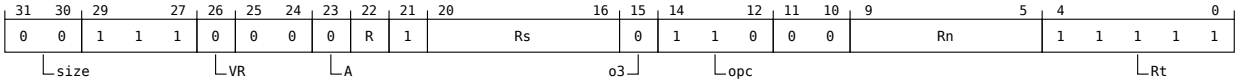
- STUMAXB does not have release semantics.
- STUMAXLB stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB. This means:

- The encodings in this description are named to match the encodings of LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB.
- The description of LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer
(FEAT_LSE)



Encoding for the No memory ordering variant Applies when (R == 0)

```
STUMAXB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMAXB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the Release variant Applies when (R == 1)

```
STUMAXLB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMAXLB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<**Ws**> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB gives the operational pseudocode for this instruction.

Operational Information

The description of LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.273 STUMAXH, STUMAXLH

Atomic unsigned maximum on halfword, without return

This instruction atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers.

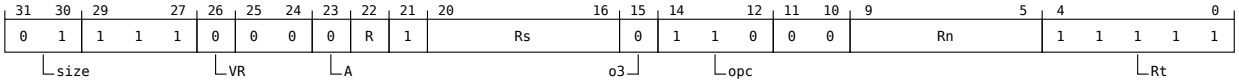
- STUMAXH does not have release semantics.
- STUMAXLH stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH. This means:

- The encodings in this description are named to match the encodings of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH.
- The description of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer
(FEAT_LSE)



Encoding for the No memory ordering variant Applies when (R == 0)

```
STUMAXH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMAXH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the Release variant Applies when (R == 1)

```
STUMAXLH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMAXLH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<**Ws**> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH gives the operational pseudocode for this instruction.

Operational Information

The description of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.274 STUMIN, STUMINL

Atomic unsigned minimum on word or doubleword, without return

This instruction atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers.

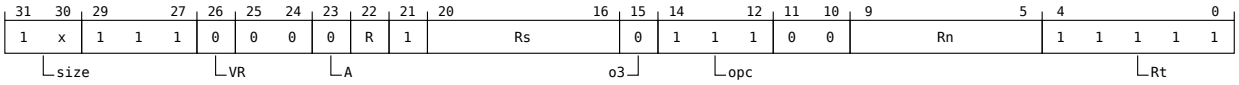
- `STUMIN` does not have release semantics.
- `STUMINL` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDUMIN, LDUMINA, LDUMINAL, LDUMINL. This means:

- The encodings in this description are named to match the encodings of LDUMIN, LDUMINA, LDUMINAL, LDUMINL.
- The description of LDUMIN, LDUMINA, LDUMINAL, LDUMINL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer
(FEAT_LSE)



Encoding for the 32-bit no memory ordering variant Applies when `(size == 10 && R == 0)`

```
STUMIN <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMIN <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 32-bit release variant Applies when `(size == 10 && R == 1)`

```
STUMINL <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMINL <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 64-bit no memory ordering variant Applies when `(size == 11 && R == 0)`

```
STUMIN <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDUMIN <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 64-bit release variant Applies when `(size == 11 && R == 1)`

```
STUMINL <Xs>, [<Xn|SP>]
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

is equivalent to

```
LDUMINL <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<**Ws**> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<**Xn**|**SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<**Xs**> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

Operation

The description of LDUMIN, LDUMINA, LDUMINAL, LDUMINL gives the operational pseudocode for this instruction.

Operational Information

The description of LDUMIN, LDUMINA, LDUMINAL, LDUMINL gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.275 STUMINB, STUMINLB

Atomic unsigned minimum on byte, without return

This instruction atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers.

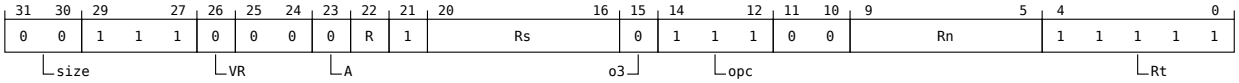
- STUMINB does not have release semantics.
- STUMINLB stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB. This means:

- The encodings in this description are named to match the encodings of LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB.
- The description of LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer
(FEAT_LSE)



Encoding for the No memory ordering variant Applies when (R == 0)

```
STUMINB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMINB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the Release variant Applies when (R == 1)

```
STUMINLB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMINLB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<**Ws**> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB gives the operational pseudocode for this instruction.

Operational Information

The description of LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.276 STUMINH, STUMINLH

Atomic unsigned minimum on halfword, without return

This instruction atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers.

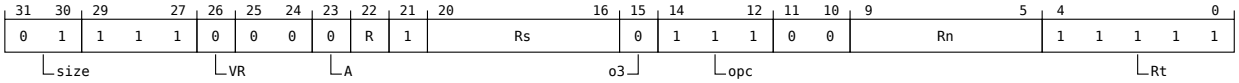
- STUMINH does not have release semantics.
- STUMINLH stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH. This means:

- The encodings in this description are named to match the encodings of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH.
- The description of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer
(FEAT_LSE)



Encoding for the No memory ordering variant Applies when (R == 0)

```
STUMINH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMINH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the Release variant Applies when (R == 1)

```
STUMINLH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMINLH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<**Ws**> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH gives the operational pseudocode for this instruction.

Operational Information

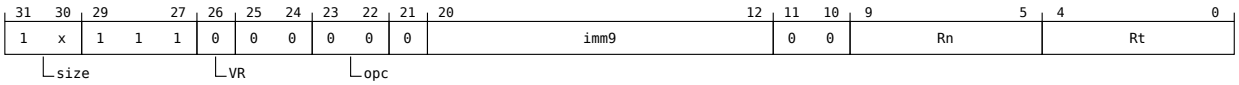
The description of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.277 STUR

Store register (unscaled)

This instruction calculates an address from a base register value and an immediate offset, and stores a 32-bit word or a 64-bit doubleword to the calculated address, from a register. For information about addressing modes, see Load/Store addressing modes.



Encoding for the 32-bit variant Applies when `(size == 10)`

```
STUR <Wt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 64-bit variant Applies when `(size == 11)`

```
STUR <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
constant integer scale = UInt(size);
constant bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<**Wt**> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<**simm**> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

<**Xt**> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

Shared Decode

```
constant integer n = UInt(Rn);
constant integer t = UInt(Rt);
constant integer datasize = 8 << scale;
constant boolean nontemporal = FALSE;
```

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

Mem[address, datasize DIV 8, accdesc] = X[t, datasize];
```

Operational information

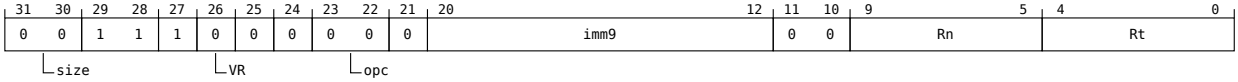
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.278 STURB

Store register byte (unscaled)

This instruction calculates an address from a base register value and an immediate offset, and stores a byte to the calculated address, from a 32-bit register. For information about addressing modes, see Load/Store addressing modes.



Encoding

```
STURB <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
constant bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<**Wt**> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<**simm**> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
constant integer n = UInt(Rn);
constant integer t = UInt(Rt);
constant integer datasize = 8;
constant boolean nontemporal = FALSE;
```

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

Mem[address, datasize DIV 8, accdesc] = X[t, datasize];
```

Operational information

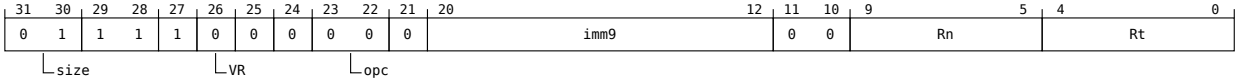
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.279 STURH

Store register halfword (unscaled)

This instruction calculates an address from a base register value and an immediate offset, and stores a halfword to the calculated address, from a 32-bit register. For information about addressing modes, see Load/Store addressing modes.



Encoding

```
STURH <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
constant bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<**Wt**> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<**simm**> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
constant integer n = UInt(Rn);
constant integer t = UInt(Rt);
constant integer datasize = 16;
constant boolean nontemporal = FALSE;
```

Operation

```
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

Mem[address, datasize DIV 8, accdesc] = X[t, datasize];
```

Operational information

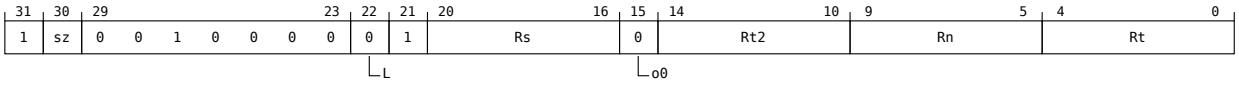
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.280 STXP

Store exclusive pair of registers

This instruction stores two 32-bit words or two 64-bit doublewords from two registers to a memory location if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See Synchronization and semaphores. For information on single-copy atomicity and alignment requirements, see Requirements for single-copy atomicity and Alignment of data accesses. If a 64-bit pair Store-Exclusive succeeds, it causes a single-copy atomic update of the 128-bit memory location being updated. For information about addressing modes, see Load/Store addressing modes.



Encoding for the 32-bit variant Applies when `(sz == 0)`

```
STXP <Ws>, <Wt1>, <Wt2>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit variant Applies when `(sz == 1)`

```
STXP <Ws>, <Xt1>, <Xt2>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer t2 = UInt(Rt2);
constant integer n = UInt(Rn);

constant integer elsize = 32 << UInt(sz);
constant integer datasize = elsize * 2;
constant boolean acqrel = FALSE;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t || (s == t2) then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
if s == n && n != 31 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly STXP.

Assembler Symbols

<**Ws**> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0 If the operation updates memory.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

1 If the operation fails to update memory.

<**Wt1**> Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<**Wt2**> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<**Xt1**> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<**Xt2**> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
bits(64) address;
bits(datasize) data;

constant integer dbytes = datasize DIV 8;
constant boolean privileged = PSTATE.EL != EL0;
constant boolean ispair = FALSE;
constant AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_STORE, acqrel, privileged, ispair, t,
                                                        t2);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n, 64];

if rt_unknown then
    data = bits(datasize) UNKNOWN;
else
    constant bits(datasize DIV 2) el1 = X[t, datasize DIV 2];
    constant bits(datasize DIV 2) el2 = X[t2, datasize DIV 2];
    data = if BigEndian(accdesc.acctype) then el1:el2 else el2:el1;
bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
// If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address,
// if accessed, would generate a synchronous Data Abort exception, it is
// IMPLEMENTATION DEFINED whether the exception is generated.
// It is a limitation of this model that synchronous Data Aborts are never
// generated in this case, as Mem[] is not called.

if AArch64.ExclusiveMonitorsPass(address, dbytes, accdesc) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    Mem[address, dbytes, accdesc] = data;
    status = ExclusiveMonitorsStatus();
X[s, 32] = ZeroExtend(status, 32);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

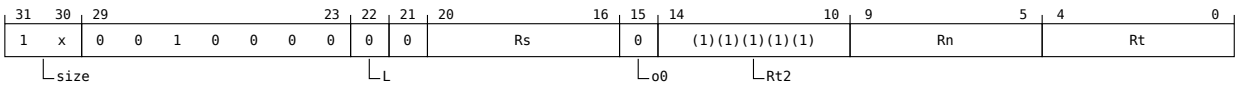
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.281 STXR

Store exclusive register

This instruction stores a 32-bit word or a 64-bit doubleword from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See Synchronization and semaphores.

For information about addressing modes, see Load/Store addressing modes.



Encoding for the 32-bit variant Applies when `(size == 10)`

```
STXR <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Encoding for the 64-bit variant Applies when `(size == 11)`

```
STXR <Ws>, <Xt>, [<Xn|SP>{, #0}]
```

Decode for all variants of this encoding

```
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer elsize = 8 << UInt(size);
constant boolean acqrel = FALSE;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
if s == n && n != 31 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly STXR.

Assembler Symbols

<**Ws**> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0 If the operation updates memory.

1 If the operation fails to update memory.

<**Wt**> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<**Xt**> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
bits(64) address;
bits(elsize) data;

constant integer dbytes = elsize DIV 8;
constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_STORE, acqrel, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n, 64];

if rt_unknown then
    data = bits(elsize) UNKNOWN;
else
    data = X[t, elsize];
bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
// If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address,
// if accessed, would generate a synchronous Data Abort exception, it is
// IMPLEMENTATION DEFINED whether the exception is generated.
// It is a limitation of this model that synchronous Data Aborts are never
// generated in this case, as Mem[] is not called.

if AArch64.ExclusiveMonitorsPass(address, dbytes, accdesc) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, dbytes, accdesc] = data;
    status = ExclusiveMonitorsStatus();
X[s, 32] = ZeroExtend(status, 32);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

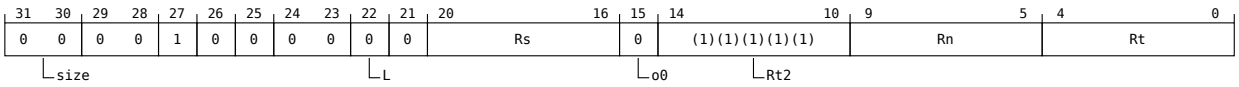
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.282 STXRB

Store exclusive register byte

This instruction stores a byte from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See Synchronization and semaphores. The memory access is atomic.

For information about addressing modes, see Load/Store addressing modes.



Encoding

```
STXRB <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Decode for this encoding

```
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acqrel = FALSE;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
if s == n && n != 31 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly STXRB.

Assembler Symbols

<**Ws**> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0 If the operation updates memory.

1 If the operation fails to update memory.

<**Wt**> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts

If a synchronous Data Abort exception is generated by the execution of this instruction:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Memory is not updated.
- <Ws> is not updated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
bits(64) address;
bits(8) data;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_STORE, acqrel, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n, 64];

if rt_unknown then
    data = bits(8) UNKNOWN;
else
    data = X[t, 8];
bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
// If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address,
// if accessed, would generate a synchronous Data Abort exception, it is
// IMPLEMENTATION DEFINED whether the exception is generated.
// It is a limitation of this model that synchronous Data Aborts are never
// generated in this case, as Mem[] is not called.

if AArch64.ExclusiveMonitorsPass(address, 1, accdesc) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, 1, accdesc] = data;
    status = ExclusiveMonitorsStatus();
X[s, 32] = ZeroExtend(status, 32);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

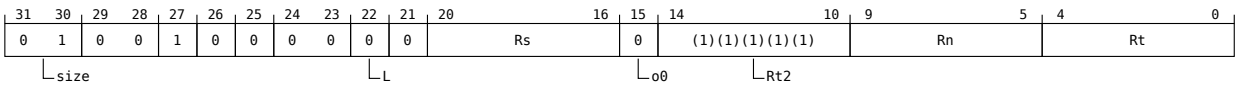
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.283 STXRH

Store exclusive register halfword

This instruction stores a halfword from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See Synchronization and semaphores. The memory access is atomic.

For information about addressing modes, see Load/Store addressing modes.



Encoding

```
STXRH <Ws>, <Wt>, [<Xn|SP>{, #0}]
```

Decode for this encoding

```
constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acqrel = FALSE;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
if s == n && n != 31 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
```

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the “Rs” field. The value returned is:

0 If the operation updates memory.

1 If the operation fails to update memory.

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- <Ws> is not updated.

A non halfword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
bits(64) address;
bits(16) data;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_STORE, acqrel, privileged, t);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n, 64];

if rt_unknown then
    data = bits(16) UNKNOWN;
else
    data = X[t, 16];
bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
// If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address,
// if accessed, would generate a synchronous Data Abort exception, it is
// IMPLEMENTATION DEFINED whether the exception is generated.
// It is a limitation of this model that synchronous Data Aborts are never
// generated in this case, as Mem[] is not called.

if AArch64.ExclusiveMonitorsPass(address, 2, accdesc) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, 2, accdesc] = data;
    status = ExclusiveMonitorsStatus();
X[s, 32] = ZeroExtend(status, 32);
```

Operational information

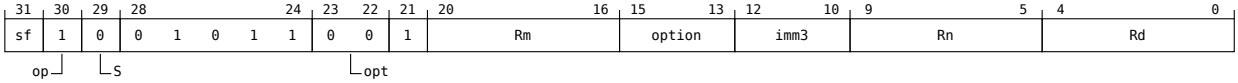
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.284 SUB (extended register)

Subtract extended and scaled register

This instruction subtracts a sign or zero-extended register value, followed by an optional left shift amount, from a register value, and writes the result to the destination register. The argument that is extended from the <Rm> register can be a byte, halfword, word, or doubleword.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
SUB <Wd|WSP>, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
SUB <Xd|SP>, <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

Decode for all variants of this encoding

```
if imm3 IN {'101', '110', '111'} then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer shift = UInt(imm3);
constant integer datasize = 32 << UInt(sf);
constant ExtendType extend_type = DecodeRegExtend(option);
```

Assembler Symbols

<Wd|WSP> Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

<Wn|WSP> Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<extend> For the "32-bit" variant: is the extension to be applied to the second source operand, encoded in "option":

option	<extend>
000	UXTB
001	UXTH
010	LSL|UXTW
011	UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

If "Rd" or "Rn" is '11111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is '010'.

For the "64-bit" variant: is the extension to be applied to the second source operand, encoded in "option":

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

option	<extend>
000	UXTB
001	UXTH
010	UXTW
011	LSL|UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

If “Rd” or “Rn” is ’11111’ (SP) and “option” is ’011’ then LSL is preferred, but may be omitted when “imm3” is ’000’. In all other cases <extend> is required and must be UXTX when “option” is ’011’.

<**amount**> Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the “imm3” field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

<**Xd|SP**> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the “Rd” field.

<**Xn|SP**> Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the “Rn” field.

<**R**> Is a width specifier, encoded in “option”:

option	<R>
00x	W
010	W
x11	X
10x	W
110	W

<**m**> Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the “Rm” field.

Operation

```
constant bits(datasize) operand1 = if n == 31 then SP[datasize] else X[n, datasize];
constant bits(datasize) operand2 = NOT(ExtendReg(m, extend_type, shift, datasize));
bits(datasize) result;

(result, -) = AddWithCarry(operand1, operand2, '1');

if d == 31 then
    SP[64] = ZeroExtend(result, 64);
else
    X[d, datasize] = result;
```

Operational information

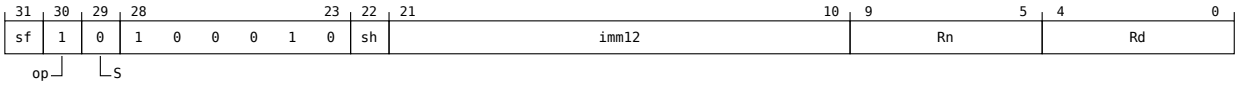
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.285 SUB (immediate)

Subtract immediate value

This instruction subtracts an optionally-shifted immediate value from a register value, and writes the result to the destination register.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
SUB <Wd|WSP>, <Wn|WSP>, #<imm>{, <shift>}
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
SUB <Xd|SP>, <Xn|SP>, #<imm>{, <shift>}
```

Decode for all variants of this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);

constant bits(24) imm = if sh == '0' then Zeros(12):imm12 else imm12:Zeros(12);
```

Assembler Symbols

<Wd|WSP> Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

<Wn|WSP> Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

<imm> Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.

<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in "sh":

sh	<shift>
0	LSL #0
1	LSL #12

<Xd|SP> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

<Xn|SP> Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

Operation

```
constant bits(datasize) operand1 = if n == 31 then SP[datasize] else X[n, datasize];
constant bits(datasize) operand2 = ZeroExtend(imm, datasize);
bits(datasize) result;
(result, -) = AddWithCarry(operand1, NOT(operand2), '1');

if d == 31 then
    SP[64] = ZeroExtend(result, 64);
else
    X[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

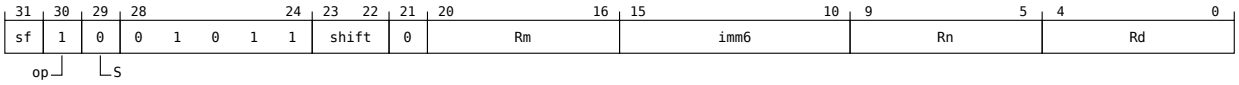
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.286 SUB (shifted register)

Subtract optionally-shifted register

This instruction subtracts an optionally-shifted register value, and writes the result to the destination register.

This instruction is used by the alias NEG (shifted register).



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
SUB <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
SUB <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

Decode for all variants of this encoding

```
if shift == '11' then EndOfDecode(Decode_UNDEF);
if sf == '0' && imm6<5> == '1' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
constant ShiftType shift_type = DecodeShift(shift);
constant integer shift_amount = UInt(imm6);
```

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Wn**> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

<**Wm**> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<**shift**> Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in "shift":

shift	<shift>
00	LSL
01	LSR
10	ASR
11	RESERVED

<**amount**> For the "32-bit" variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the "64-bit" variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Xn**> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**Xm**> Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Alias Conditions

Alias	Is preferred when
NEG (shifted register)	Rn == '11111'

Operation

```
constant bits(datasize) operand1 = X[n, datasize];
constant bits(datasize) operand2 = NOT(ShiftReg(m, shift_type, shift_amount, datasize));
bits(datasize) result;

(result, -) = AddWithCarry(operand1, operand2, '1');

X[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

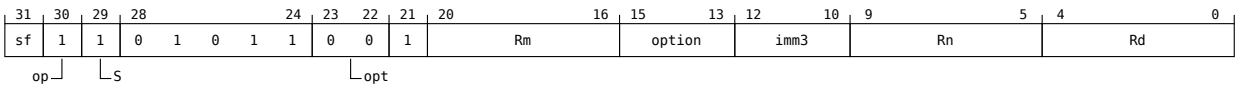
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.287 SUBS (extended register)

Subtract extended and scaled register, setting flags

This instruction subtracts a sign or zero-extended register value, followed by an optional left shift amount, from a register value, and writes the result to the destination register. The argument that is extended from the <Rm> register can be a byte, halfword, word, or doubleword. It updates the condition flags based on the result.

This instruction is used by the alias CMP (extended register).



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
SUBS <Wd>, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
SUBS <Xd>, <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

Decode for all variants of this encoding

```
if imm3 IN {'101', '110', '111'} then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer shift = UInt(imm3);
constant integer datasize = 32 << UInt(sf);
constant ExtendType extend_type = DecodeRegExtend(option);
```

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn|WSP> Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<extend> For the "32-bit" variant: is the extension to be applied to the second source operand, encoded in "option":

option	<extend>
000	UXTB
001	UXTH
010	LSL|UXTW
011	UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

If "Rn" is '11111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is '010'.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For the “64-bit” variant: is the extension to be applied to the second source operand, encoded in “option”:

option	<extend>
000	UXTB
001	UXTH
010	UXTW
011	LSL|UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

If “Rn” is ’11111’ (SP) and “option” is ’011’ then LSL is preferred, but may be omitted when “imm3” is ’000’. In all other cases <extend> is required and must be UXTX when “option” is ’011’.

<**amount**> Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the “imm3” field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Xn|SP**> Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the “Rn” field.

<**R**> Is a width specifier, encoded in “option”:

option	<R>
00x	W
010	W
x11	X
10x	W
110	W

<**m**> Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the “Rm” field.

Alias Conditions

Alias	Is preferred when
CMP (extended register)	Rd == '11111'

Operation

```
constant bits(datasize) operand1 = if n == 31 then SP[datasize] else X[n, datasize];
constant bits(datasize) operand2 = NOT(ExtendReg(m, extend_type, shift, datasize));
bits(datasize) result;
bits(4) nzcv;

(result, nzcv) = AddWithCarry(operand1, operand2, '1');

X[d, datasize] = result;
PSTATE.<N,Z,C,V> = nzcv;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

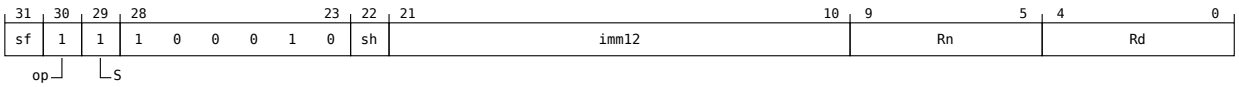
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.288 SUBS (immediate)

Subtract immediate value, setting flags

This instruction subtracts an optionally-shifted immediate value from a register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias CMP (immediate).



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
SUBS <Wd>, <Wn|WSP>, #<imm>{, <shift>}
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
SUBS <Xd>, <Xn|SP>, #<imm>{, <shift>}
```

Decode for all variants of this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);

constant bits(24) imm = if sh == '0' then Zeros(12):imm12 else imm12:Zeros(12);
```

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Wn|WSP**> Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

<**imm**> Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.

<**shift**> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in "sh":

sh	<shift>
0	LSL #0
1	LSL #12

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Xn|SP**> Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
CMP (immediate)	Rd == '11111'

Operation

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant bits(datasize) operand1 = if n == 31 then SP[datasize] else X[n, datasize];
constant bits(datasize) operand2 = ZeroExtend(imm, datasize);
bits(datasize) result;
bits(4) nzcv;

(result, nzcv) = AddWithCarry(operand1, NOT(operand2), '1');

X[d, datasize] = result;
PSTATE.<N,Z,C,V> = nzcv;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

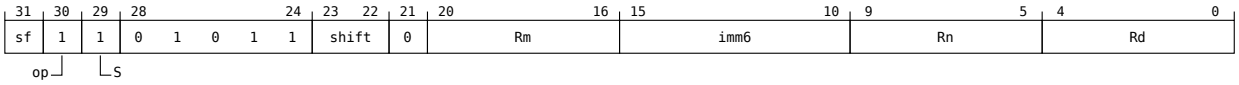
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.289 SUBS (shifted register)

Subtract optionally-shifted register, setting flags

This instruction subtracts an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the aliases CMP (shifted register), and NEGS.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
SUBS <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
SUBS <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

Decode for all variants of this encoding

```
if shift == '11' then EndOfDecode(Decode_UNDEF);
if sf == '0' && imm6<5> == '1' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
constant ShiftType shift_type = DecodeShift(shift);
constant integer shift_amount = UInt(imm6);
```

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Wn**> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

<**Wm**> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<**shift**> Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in "shift":

shift	<shift>
00	LSL
01	LSR
10	ASR
11	RESERVED

<**amount**> For the "32-bit" variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the "64-bit" variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Xn**> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**Xm**> Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Alias Conditions

Alias	Is preferred when
CMP (shifted register)	Rd == '11111'
NEGS	Rn == '11111' && Rd != '11111'

Operation

```
constant bits(datasize) operand1 = X[n, datasize];
constant bits(datasize) operand2 = NOT(ShiftReg(m, shift_type, shift_amount, datasize));
bits(datasize) result;
bits(4) nzcv;

(result, nzcv) = AddWithCarry(operand1, operand2, '1');

X[d, datasize] = result;
PSTATE.<N,Z,C,V> = nzcv;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

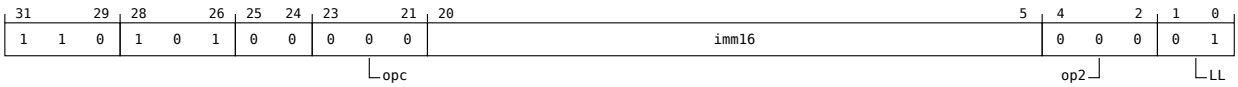
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.290 SVC

Supervisor call

This instruction causes an exception to be taken to EL1.

On executing an `svc` instruction, the PE records the exception as a Supervisor Call exception in ESR_ELx, using the EC syndrome value 0x15, and the value of the immediate argument.



Encoding

```
SVC #<imm>
```

Decode for this encoding

```
constant bits(16) imm = imm16;
```

Assembler Symbols

<imm> Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

Operation

```
AArch64.CallSupervisor(imm);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.291 SWP, SWPA, SWPAL, SWPL

Swap word or doubleword in memory

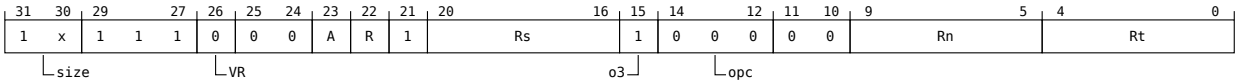
This instruction atomically loads a 32-bit word or 64-bit doubleword from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, `SWPA` and `SWPAL` load from memory with acquire semantics.
- `SWPL` and `SWPAL` store to memory with release semantics.
- `SWP` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Integer
(FEAT_LSE)



Encoding for the 32-bit SWP variant Applies when `(size == 10 && A == 0 && R == 0)`

```
SWP <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit SWPA variant Applies when `(size == 10 && A == 1 && R == 0)`

```
SWPA <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit SWPAL variant Applies when `(size == 10 && A == 1 && R == 1)`

```
SWPAL <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 32-bit SWPL variant Applies when `(size == 10 && A == 0 && R == 1)`

```
SWPL <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the 64-bit SWP variant Applies when `(size == 11 && A == 0 && R == 0)`

```
SWP <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit SWPA variant Applies when `(size == 11 && A == 1 && R == 0)`

```
SWPA <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit SWPAL variant Applies when `(size == 11 && A == 1 && R == 1)`

```
SWPAL <Xs>, <Xt>, [<Xn|SP>]
```

Encoding for the 64-bit SWPL variant Applies when `(size == 11 && A == 0 && R == 1)`

```
SWPL <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);

constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant integer datasize = 8 << UInt(size);
constant integer regsize = if datasize == 64 then 64 else 32;
constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
```

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register to be stored, encoded in the “Rs” field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Xs> Is the 64-bit name of the general-purpose register to be stored, encoded in the “Rs” field.

<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

Operation

```
bits(64) address;
bits(datasize) data;
bits(datasize) store_value;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_SWP, acquire, release,
                                                          privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

store_value = X[s, datasize];

constant bits(datasize) comparevalue = bits(datasize) UNKNOWN; // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, store_value, accdesc);

X[t, regsize] = ZeroExtend(data, regsize);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.292 SWPB, SWPAB, SWPALB, SWPLB

Swap byte in memory

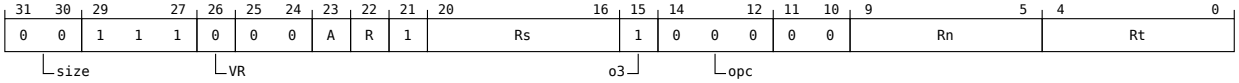
This instruction atomically loads an 8-bit byte from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, `SWPAB` and `SWPALB` load from memory with acquire semantics.
- `SWPLB` and `SWPALB` store to memory with release semantics.
- `SWPB` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Integer
(FEAT_LSE)



Encoding for the SWPB variant Applies when `(A == 0 && R == 0)`

```
SWPB <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the SWPAB variant Applies when `(A == 1 && R == 0)`

```
SWPAB <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the SWPALB variant Applies when `(A == 1 && R == 1)`

```
SWPALB <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the SWPLB variant Applies when `(A == 0 && R == 1)`

```
SWPLB <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);

constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
```

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register to be stored, encoded in the “Rs” field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
bits(64) address;
bits(8) data;
bits(8) store_value;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_SWP, acquire, release,
                                                          privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

store_value = X[s, 8];

constant bits(8) comparevalue = bits(8) UNKNOWN; // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, store_value, accdesc);

X[t, 32] = ZeroExtend(data, 32);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.293 SWPH, SWPAH, SWPALH, SWPLH

Swap halfword in memory

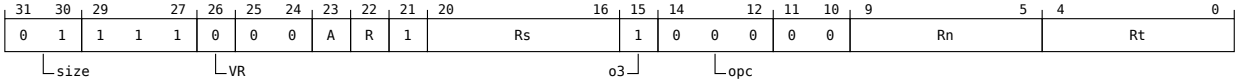
This instruction atomically loads a 16-bit halfword from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, `SWPAH` and `SWPALH` load from memory with acquire semantics.
- `SWPLH` and `SWPALH` store to memory with release semantics.
- `SWPH` has neither acquire nor release semantics.

For more information about memory ordering semantics, see Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

Integer
(FEAT_LSE)



Encoding for the SWPH variant Applies when `(A == 0 && R == 0)`

```
SWPH <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the SWPAH variant Applies when `(A == 1 && R == 0)`

```
SWPAH <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the SWPALH variant Applies when `(A == 1 && R == 1)`

```
SWPALH <Ws>, <Wt>, [<Xn|SP>]
```

Encoding for the SWPLH variant Applies when `(A == 0 && R == 1)`

```
SWPLH <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then EndOfDecode(Decode_UNDEF);

constant integer s = UInt(Rs);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);

constant boolean acquire = A == '1' && Rt != '11111';
constant boolean release = R == '1';
```

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register to be stored, encoded in the “Rs” field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

Operation

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
bits(64) address;
bits(16) data;
bits(16) store_value;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_SWP, acquire, release,
                                                          privileged, t, s);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

store_value = X[s, 16];

constant bits(16) comparevalue = bits(16) UNKNOWN; // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, store_value, accdesc);

X[t, 32] = ZeroExtend(data, 32);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

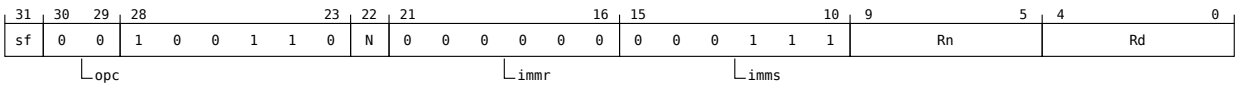
C6.2.294 SXTB

Signed extend byte

This instruction extracts an 8-bit value from a register, sign-extends it to the size of the register, and writes the result to the destination register.

This is an alias of SBFM. This means:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding for the 32-bit variant Applies when `(sf == 0 && N == 0)`

```
SXTB <Wd>, <Wn>
```

is equivalent to

```
SBFM <Wd>, <Wn>, #0, #7
```

and is always the preferred disassembly.

Encoding for the 64-bit variant Applies when `(sf == 1 && N == 1)`

```
SXTB <Xd>, <Wn>
```

is equivalent to

```
SBFM <Xd>, <Xn>, #0, #7
```

and is always the preferred disassembly.

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wn**> Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Operational Information

The description of SBFM gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

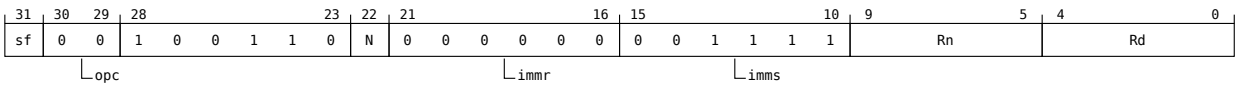
C6.2.295 SXTH

Sign extend halfword

This instruction extracts a 16-bit value, sign-extends it to the size of the register, and writes the result to the destination register.

This is an alias of SBFM. This means:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding for the 32-bit variant Applies when `(sf == 0 && N == 0)`

```
SXTH <Wd>, <Wn>
```

is equivalent to

```
SBFM <Wd>, <Wn>, #0, #15
```

and is always the preferred disassembly.

Encoding for the 64-bit variant Applies when `(sf == 1 && N == 1)`

```
SXTH <Xd>, <Wn>
```

is equivalent to

```
SBFM <Xd>, <Xn>, #0, #15
```

and is always the preferred disassembly.

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wn**> Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Operational Information

The description of SBFM gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

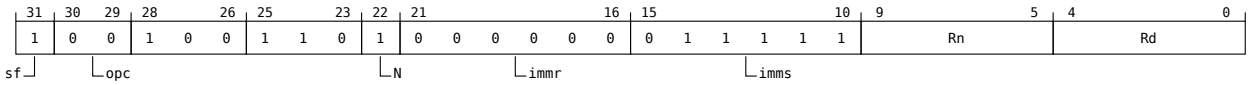
C6.2.296 SXTW

Sign extend word

This instruction sign-extends a word to the size of the register, and writes the result to the destination register.

This is an alias of SBFM. This means:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding

SXTW <Xd>, <Wn>

is equivalent to

SBFM <Xd>, <Xn>, #0, #31

and is always the preferred disassembly.

Assembler Symbols

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wn**> Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Operational Information

The description of SBFM gives the operational information for this instruction.

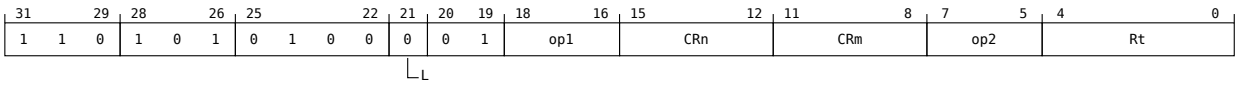
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.297 SYS

System instruction

For more information, see Op0 equals 0b01, cache maintenance, TLB maintenance, and address translation instructions for the encodings of System instructions.

This instruction is used by the aliases AT, CFP, CPP, DC, DVP, IC, and TLBI.



Encoding

```
SYS #<op1>, <Cn>, <Cm>, #<op2>{, <Xt>}
```

Decode for this encoding

```
constant integer t       = UInt(Rt);
constant bits(1) sys_L   = L;
constant bits(2) sys_op0 = '01';
constant bits(3) sys_op1 = op1;
constant bits(3) sys_op2 = op2;
constant bits(4) sys_crn = CRn;
constant bits(4) sys_crm = CRm;
```

Assembler Symbols

<**op1**> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the “op1” field.

<**Cn**> Is a name ’Cn’, with ’n’ in the range 0 to 15, encoded in the “CRn” field.

<**Cm**> Is a name ’Cm’, with ’m’ in the range 0 to 15, encoded in the “CRm” field.

<**op2**> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the “op2” field.

<**Xt**> Is the 64-bit name of the optional general-purpose source register, defaulting to ’11111’, encoded in the “Rt” field.

Alias Conditions

Alias	Is preferred when
AT	`CRn == '0111' && CRm IN {'100x'} &&SysOp(op1, '0111', CRm, op2) == Sys_AT`
CFP	`op1 == '011' && CRn == '0111' && CRm == '0011' && op2 == '100'`
CPP	`op1 == '011' && CRn == '0111' && CRm == '0011' && op2 == '111'`
DC	`CRn == '0111' &&SysOp(op1, '0111', CRm, op2) == Sys_DC`
DVP	`op1 == '011' && CRn == '0111' && CRm == '0011' && op2 == '101'`
IC	`CRn == '0111' &&SysOp(op1, '0111', CRm, op2) == Sys_IC`

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Alias	Is preferred when
TLBI	CRn IN {'100x'} &&SysOp(op1, CRn, CRm, op2) == Sys_TLBI

Operation

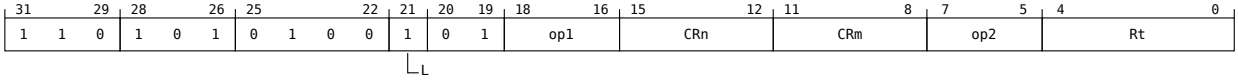
```
AArch64.SysInstr(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2, t);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.298 SYSL

System instruction with result

For more information, see Op0 equals 0b01, cache maintenance, TLB maintenance, and address translation instructions for the encodings of System instructions.



Encoding

```
SYSL <Xt>, #<op1>, <Cn>, <Cm>, #<op2>
```

Decode for this encoding

```
constant integer t       = UInt(Rt);
constant bits(1) sys_L   = L;
constant bits(2) sys_op0 = '01';
constant bits(3) sys_op1 = op1;
constant bits(3) sys_op2 = op2;
constant bits(4) sys_crn = CRn;
constant bits(4) sys_crm = CRm;
```

Assembler Symbols

<**Xt**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rt” field.

<**op1**> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the “op1” field.

<**Cn**> Is a name ’Cn’, with ’n’ in the range 0 to 15, encoded in the “CRn” field.

<**Cm**> Is a name ’Cm’, with ’m’ in the range 0 to 15, encoded in the “CRm” field.

<**op2**> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the “op2” field.

Operation

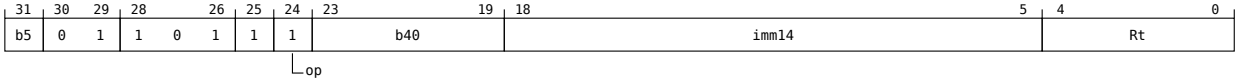
```
AArch64.SysInstrWithResult(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2, t);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.299 TBNZ

Test bit and branch if nonzero

This instruction compares the value of a bit in a general-purpose register with zero, and conditionally branches to a label at a PC-relative offset if the comparison is not equal. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect condition flags.



Encoding

```
TBNZ <R><t>, #<imm>, <label>
```

Decode for this encoding

```
constant integer t = UInt(Rt);

constant integer datasize = 32 << UInt(b5);
constant integer bit_pos = UInt(b5:b40);
constant bits(64) offset = SignExtend(imm14:'00', 64);
```

Assembler Symbols

<**R**> Is a width specifier, encoded in “b5”:

b5	<R>
0	W
1	X

In assembler source code an ’X’ specifier is always permitted, but a ’W’ specifier is only permitted when the bit number is less than 32.

<**t**> Is the number [0-30] of the general-purpose register to be tested or the name ZR (31), encoded in the “Rt” field.

<**imm**> Is the bit number to be tested, in the range 0 to 63, encoded in “b5:b40”.

<**label**> Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-32KB, is encoded as “imm14” times 4.

Operation

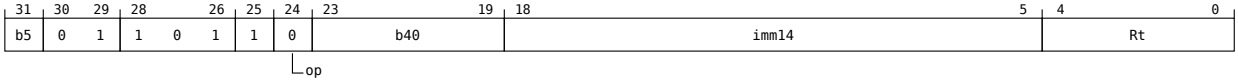
```
constant bits(datasize) operand = X[t, datasize];
constant boolean branch_conditional = TRUE;
if operand<bit_pos> != '0' then
    BranchTo(PC64 + offset, BranchType_DIR, branch_conditional);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.300 TBZ

Test bit and branch if zero

This instruction compares the value of a test bit with zero, and conditionally branches to a label at a PC-relative offset if the comparison is equal. This instruction provides a hint that this is not a subroutine call or return. This instruction does not affect condition flags.



Encoding

```
TBZ <R><t>, #<imm>, <label>
```

Decode for this encoding

```
constant integer t = UInt(Rt);

constant integer datasize = 32 << UInt(b5);
constant integer bit_pos = UInt(b5:b40);
constant bits(64) offset = SignExtend(imm14:'00', 64);
```

Assembler Symbols

<**R**> Is a width specifier, encoded in “b5”:

b5	<R>
0	W
1	X

In assembler source code an ’X’ specifier is always permitted, but a ’W’ specifier is only permitted when the bit number is less than 32.

<**t**> Is the number [0-30] of the general-purpose register to be tested or the name ZR (31), encoded in the “Rt” field.

<**imm**> Is the bit number to be tested, in the range 0 to 63, encoded in “b5:b40”.

<**label**> Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-32KB, is encoded as “imm14” times 4.

Operation

```
constant bits(datasize) operand = X[t, datasize];
constant boolean branch_conditional = TRUE;
if operand<bit_pos> == '0' then
    BranchTo(PC64 + offset, BranchType_DIR, branch_conditional);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

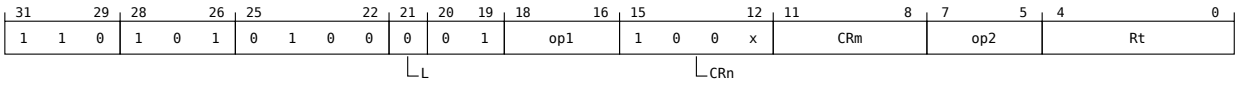
C6.2.301 TLBI

TLB invalidate operation

For more information, see op0==0b01, cache maintenance, TLB maintenance, and address translation instructions.

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding

```
TLBI <tlbi_op>{, <Xt>}
```

is equivalent to

```
SYS #<op1>, <Cn>, <Cm>, #<op2>{, <Xt>}
```

and is the preferred disassembly when `SysOp(op1, CRn, CRm, op2) == Sys_TLBI`.

Assembler Symbols

<**tlbi_op**> Is a TLBI operation name, as listed for the TLBI system instruction group, encoded in "op1:CRn:CRm:op2":

op1	CRn	CRm	op2	<tlbi_op>	Architectural Feature
000	1000	0001	000	VMALLE1OS	FEAT_TLBIOS
000	1000	0001	001	VAE1OS	FEAT_TLBIOS
000	1000	0001	010	ASIDE1OS	FEAT_TLBIOS
000	1000	0001	011	VAAE1OS	FEAT_TLBIOS
000	1000	0001	101	VALE1OS	FEAT_TLBIOS
000	1000	0001	111	VAALE1OS	FEAT_TLBIOS
000	1000	0010	001	RVAE1IS	FEAT_TLBIRANGE
000	1000	0010	011	RVAAE1IS	FEAT_TLBIRANGE
000	1000	0010	101	RVALE1IS	FEAT_TLBIRANGE
000	1000	0010	111	RVAALE1IS	FEAT_TLBIRANGE
000	1000	0011	000	VMALLE1IS	-
000	1000	0011	001	VAE1IS	-
000	1000	0011	010	ASIDE1IS	-
000	1000	0011	011	VAAE1IS	-
000	1000	0011	101	VALE1IS	-
000	1000	0011	111	VAALE1IS	-
000	1000	0101	001	RVAE1OS	FEAT_TLBIRANGE

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	<tlbi_op>	Architectural Feature
000	1000	0101	011	RVAAE1OS	FEAT_TLBIRANGE
000	1000	0101	101	RVALE1OS	FEAT_TLBIRANGE
000	1000	0101	111	RVAALE1OS	FEAT_TLBIRANGE
000	1000	0110	001	RVAE1	FEAT_TLBIRANGE
000	1000	0110	011	RVAAE1	FEAT_TLBIRANGE
000	1000	0110	101	RVALE1	FEAT_TLBIRANGE
000	1000	0110	111	RVAALE1	FEAT_TLBIRANGE
000	1000	0111	000	VMALLE1	-
000	1000	0111	001	VAE1	-
000	1000	0111	010	ASIDE1	-
000	1000	0111	011	VAAE1	-
000	1000	0111	101	VALE1	-
000	1000	0111	111	VAALE1	-
100	1000	0000	001	IPAS2E1IS	-
100	1000	0000	010	RIPAS2E1IS	FEAT_TLBIRANGE
100	1000	0000	101	IPAS2LE1IS	-
100	1000	0000	110	RIPAS2LE1IS	FEAT_TLBIRANGE
100	1000	0001	000	ALLE2OS	FEAT_TLBIOS
100	1000	0001	001	VAE2OS	FEAT_TLBIOS
100	1000	0001	100	ALLE1OS	FEAT_TLBIOS
100	1000	0001	101	VALE2OS	FEAT_TLBIOS
100	1000	0001	110	VMALLS12E1OS	FEAT_TLBIOS
100	1000	0010	001	RVAE2IS	FEAT_TLBIRANGE
100	1000	0010	101	RVALE2IS	FEAT_TLBIRANGE
100	1000	0011	000	ALLE2IS	-
100	1000	0011	001	VAE2IS	-
100	1000	0011	100	ALLE1IS	-
100	1000	0011	101	VALE2IS	-
100	1000	0011	110	VMALLS12E1IS	-
100	1000	0100	000	IPAS2E1OS	FEAT_TLBIOS
100	1000	0100	001	IPAS2E1	-
100	1000	0100	010	RIPAS2E1	FEAT_TLBIRANGE
100	1000	0100	011	RIPAS2E1OS	FEAT_TLBIRANGE
100	1000	0100	100	IPAS2LE1OS	FEAT_TLBIOS
100	1000	0100	101	IPAS2LE1	-
100	1000	0100	110	RIPAS2LE1	FEAT_TLBIRANGE
100	1000	0100	111	RIPAS2LE1OS	FEAT_TLBIRANGE
100	1000	0101	001	RVAE2OS	FEAT_TLBIRANGE
100	1000	0101	101	RVALE2OS	FEAT_TLBIRANGE
100	1000	0110	001	RVAE2	FEAT_TLBIRANGE
100	1000	0110	101	RVALE2	FEAT_TLBIRANGE

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	<tlbi_op>	Architectural Feature
100	1000	0111	000	ALLE2	-
100	1000	0111	001	VAE2	-
100	1000	0111	100	ALLE1	-
100	1000	0111	101	VALE2	-
100	1000	0111	110	VMALLS12E1	-
110	1000	0001	000	ALLE3OS	FEAT_TLBIOS
110	1000	0001	001	VAE3OS	FEAT_TLBIOS
110	1000	0001	101	VALE3OS	FEAT_TLBIOS
110	1000	0010	001	RVAE3IS	FEAT_TLBIRANGE
110	1000	0010	101	RVALE3IS	FEAT_TLBIRANGE
110	1000	0011	000	ALLE3IS	-
110	1000	0011	001	VAE3IS	-
110	1000	0011	101	VALE3IS	-
110	1000	0101	001	RVAE3OS	FEAT_TLBIRANGE
110	1000	0101	101	RVALE3OS	FEAT_TLBIRANGE
110	1000	0110	001	RVAE3	FEAT_TLBIRANGE
110	1000	0110	101	RVALE3	FEAT_TLBIRANGE
110	1000	0111	000	ALLE3	-
110	1000	0111	001	VAE3	-
110	1000	0111	101	VALE3	-

<**Xt**> Is the 64-bit name of the optional general-purpose source register, defaulting to ’11111’, encoded in the “Rt” field.

Operation

The description of SYS gives the operational pseudocode for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

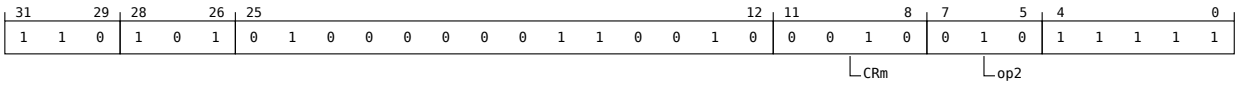
C6.2.302 TSB

Trace synchronization barrier

This instruction is a barrier that synchronizes the trace operations of instructions, see Trace Synchronization Barrier (TSB).

If FEAT_TRF is not implemented, this instruction executes as a `NOP`.

System
(FEAT_TRF)



Encoding

```
TSB CSYNC
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_TRF) then EndOfDecode(Decode_NOP);
```

Operation

```
TraceSynchronizationBarrier();
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

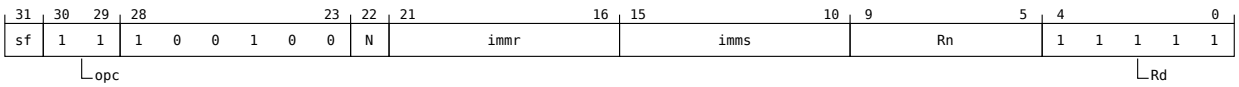
C6.2.303 TST (immediate)

Test bits (immediate)

This instruction performs a bitwise AND of a register value and an immediate value, and discards the results. It updates the condition flags based on the result.

This is an alias of ANDS (immediate). This means:

- The encodings in this description are named to match the encodings of ANDS (immediate).
- The description of ANDS (immediate) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding for the 32-bit variant Applies when `(sf == 0 && N == 0)`

```
TST <Wn>, #<imm>
```

is equivalent to

```
ANDS WZR, <Wn>, #<imm>
```

and is always the preferred disassembly.

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
TST <Xn>, #<imm>
```

is equivalent to

```
ANDS XZR, <Xn>, #<imm>
```

and is always the preferred disassembly.

Assembler Symbols

<**Wn**> Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<**imm**> For the “32-bit” variant: is the bitmask immediate, encoded in “imms:immr”.

For the “64-bit” variant: is the bitmask immediate, encoded in “N:imms:immr”.

<**Xn**> Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

The description of ANDS (immediate) gives the operational pseudocode for this instruction.

Operational Information

The description of ANDS (immediate) gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

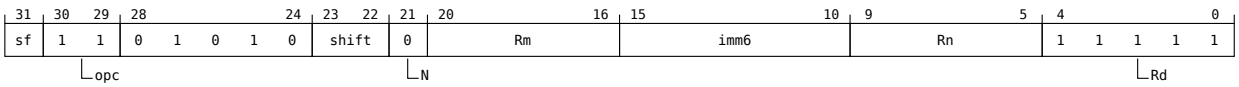
C6.2.304 TST (shifted register)

Test (shifted register)

This instruction performs a bitwise AND operation on a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

This is an alias of ANDS (shifted register). This means:

- The encodings in this description are named to match the encodings of ANDS (shifted register).
- The description of ANDS (shifted register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
TST <Wn>, <Wm>{, <shift> #<amount>}
```

is equivalent to

```
ANDS WZR, <Wn>, <Wm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
TST <Xn>, <Xm>{, <shift> #<amount>}
```

is equivalent to

```
ANDS XZR, <Xn>, <Xm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

Assembler Symbols

<**Wn**> Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Wm**> Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<**shift**> Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:

shift	<**shift**>
00	LSL
01	LSR
10	ASR
11	ROR

<**amount**> For the “32-bit” variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the “imm6” field.

For the “64-bit” variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the “imm6” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Operation

The description of ANDS (shifted register) gives the operational pseudocode for this instruction.

Operational Information

The description of ANDS (shifted register) gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

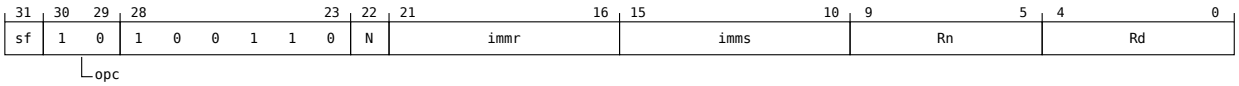
C6.2.305 UBFIZ

Unsigned bitfield insert in zeros

This instruction copies a bitfield of <width> bits from the least significant bits of the source register to bit position <lsb> of the destination register, setting the destination bits above and below the bitfield to zero.

This is an alias of UBFM. This means:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding for the 32-bit variant Applies when `(sf == 0 && N == 0)`

```
UBFIZ <Wd>, <Wn>, #<lsb>, #<width>
```

is equivalent to

```
UBFM <Wd>, <Wn>, #(-<lsb> MOD 32), #(<width>-1)
```

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

Encoding for the 64-bit variant Applies when `(sf == 1 && N == 1)`

```
UBFIZ <Xd>, <Xn>, #<lsb>, #<width>
```

is equivalent to

```
UBFM <Xd>, <Xn>, #(-<lsb> MOD 64), #(<width>-1)
```

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Wn**> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

<**lsb**> For the "32-bit" variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 31.

For the "64-bit" variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 63.

<**width**> For the "32-bit" variant: is the width of the bitfield, in the range 1 to 32-<lsb>.

For the "64-bit" variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Xn**> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of UBFM gives the operational pseudocode for this instruction.

Operational Information

The description of UBFM gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.306 UBFM

Unsigned bitfield move

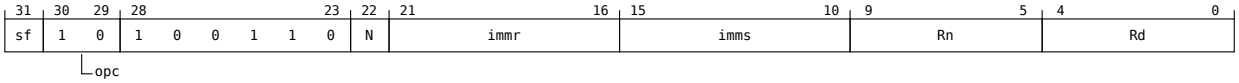
This instruction is usually accessed via one of its aliases, which are always preferred for disassembly.

If <imms> is greater than or equal to <immr>, this copies a bitfield of (<imms>-<immr>+1) bits starting from bit position <immr> in the source register to the least significant bits of the destination register.

If <imms> is less than <immr>, this copies a bitfield of (<imms>+1) bits from the least significant bits of the source register to bit position (regsize-<immr>) of the destination register, where regsize is the destination register size of 32 or 64 bits.

In both cases, the destination bits below and above the bitfield are set to zero.

This instruction is used by the aliases LSL (immediate), LSR (immediate), UBFIZ, UBFX, UXTB, and UXTH.



Encoding for the 32-bit variant Applies when `(sf == 0 && N == 0)`

```
UBFM <Wd>, <Wn>, #<immr>, #<imms>
```

Encoding for the 64-bit variant Applies when `(sf == 1 && N == 1)`

```
UBFM <Xd>, <Xn>, #<immr>, #<imms>
```

Decode for all variants of this encoding

```
if sf == '1' && N != '1' then EndOfDecode(Decode_UNDEF);
if sf == '0' && (N != '0' || immr<5> != '0' || imms<5> != '0') then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);
constant integer s = UInt(imms);
constant integer r = UInt(immr);

bits(datasize) wmask;
bits(datasize) tmask;
(wmask, tmask) = DecodeBitMasks(N, imms, immr, FALSE, datasize);
```

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wn**> Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<**immr**> For the “32-bit” variant: is the right rotate amount, in the range 0 to 31, encoded in the “immr” field.

For the “64-bit” variant: is the right rotate amount, in the range 0 to 63, encoded in the “immr” field.

<**imms**> For the “32-bit” variant: is the leftmost bit number to be moved from the source, in the range 0 to 31, encoded in the “imms” field.

For the “64-bit” variant: is the leftmost bit number to be moved from the source, in the range 0 to 63, encoded in the “imms” field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Alias Conditions

Alias	Of variant	Is preferred when
LSL (immediate)	32-bit	`imms != '011111' &&UInt(imms) + 1 == UInt(immr)`
LSL (immediate)	64-bit	`imms != '111111' &&UInt(imms) + 1 == UInt(immr)`
LSR (immediate)	32-bit	`imms == '011111'`
LSR (immediate)	64-bit	`imms == '111111'`
UBFIZ		`UInt(imms) < UInt(immr)`
UBFX		`BFXPreferred(sf, opc<1>, imms, immr)`
UXTB		`immr == '000000' && imms == '000111'`
UXTH		`immr == '000000' && imms == '001111'`

Operation

```
constant bits(datasize) src = X[n, datasize];

// Perform bitfield move on low bits
constant bits(datasize) bot = ROR(src, r) AND wmask;

// Combine extension bits and result bits
X[d, datasize] = bot AND tmask;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

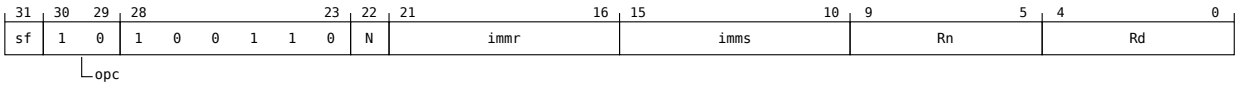
C6.2.307 UBFX

Unsigned bitfield extract

This instruction copies a bitfield of <width> bits starting from bit position <lsb> in the source register to the least significant bits of the destination register, and sets destination bits above the bitfield to zero.

This is an alias of UBFM. This means:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding for the 32-bit variant Applies when `(sf == 0 && N == 0)`

```
UBFX <Wd>, <Wn>, #<lsb>, #<width>
```

is equivalent to

```
UBFM <Wd>, <Wn>, #<lsb>, #(<lsb>+<width>-1)
```

and is the preferred disassembly when `BFXPreferred(sf, opc<1>, imms, immr)`.

Encoding for the 64-bit variant Applies when `(sf == 1 && N == 1)`

```
UBFX <Xd>, <Xn>, #<lsb>, #<width>
```

is equivalent to

```
UBFM <Xd>, <Xn>, #<lsb>, #(<lsb>+<width>-1)
```

and is the preferred disassembly when `BFXPreferred(sf, opc<1>, imms, immr)`.

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Wn**> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

<**lsb**> For the "32-bit" variant: is the bit number of the lsb of the source bitfield, in the range 0 to 31.

For the "64-bit" variant: is the bit number of the lsb of the source bitfield, in the range 0 to 63.

<**width**> For the "32-bit" variant: is the width of the bitfield, in the range 1 to 32-<lsb>.

For the "64-bit" variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Xn**> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of UBFM gives the operational pseudocode for this instruction.

Operational Information

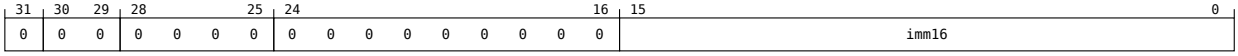
The description of UBFM gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.308 UDF

Permanently undefined

This instruction generates an Undefined Instruction exception (ESR_ELx.EC = 0b000000). The encodings for `UDF` used in this section are defined as permanently UNDEFINED.



Encoding

```
UDF #<imm>
```

Decode for this encoding

```
// The imm16 field is ignored by hardware.
EndOfDecode(Decode_UNDEF);
```

Assembler Symbols

<imm> is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field. The PE ignores the value of this constant.

Operation

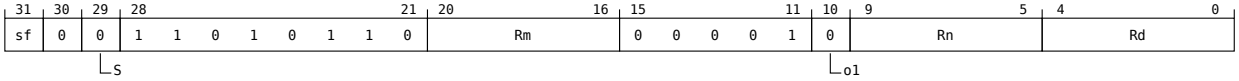
```
// No operation.
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.309 UDIV

Unsigned divide

This instruction divides the first unsigned source register value by the second unsigned source register value, and writes the result to the destination register. Dividing by zero writes the value zero to the destination register. The condition flags are not affected.



Encoding for the 32-bit variant Applies when `(sf == 0)`

```
UDIV <Wd>, <Wn>, <Wm>
```

Encoding for the 64-bit variant Applies when `(sf == 1)`

```
UDIV <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
```

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wn**> Is the 32-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Wm**> Is the 32-bit name of the second general-purpose source register, encoded in the “Rm” field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Xn**> Is the 64-bit name of the first general-purpose source register, encoded in the “Rn” field.

<**Xm**> Is the 64-bit name of the second general-purpose source register, encoded in the “Rm” field.

Operation

```
constant bits(datasize) operand1 = X[n, datasize];
constant bits(datasize) operand2 = X[m, datasize];
constant integer dividend = UInt(operand1);
constant integer divisor  = UInt(operand2);
integer result;
if divisor == 0 then
    result = 0;
else
    result = dividend DIV divisor;
X[d, datasize] = result<datasize-1:0>;
```

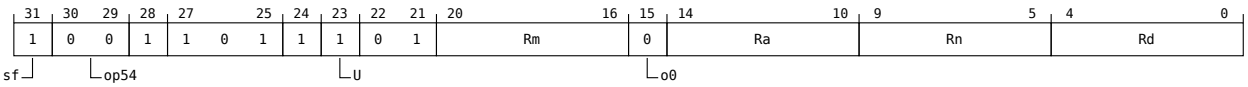

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.310 UMADDL

Unsigned multiply-add long

This instruction multiplies two 32-bit register values, adds a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias UMULL.



Encoding

UMADDL <Xd>, <Wn>, <Wm>, <Xa>

Decode for this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer a = UInt(Ra);
```

Assembler Symbols

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Wn**> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<**Wm**> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

<**Xa**> Is the 64-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.

Alias Conditions

Alias	Is preferred when
UMULL	Ra == '11111'

Operation

```
constant bits(32) operand1 = X[n, 32];
constant bits(32) operand2 = X[m, 32];
constant bits(64) operand3 = X[a, 64];

constant integer result = UInt(operand3) + (UInt(operand1) * UInt(operand2));

X[d, 64] = result<63:0>;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

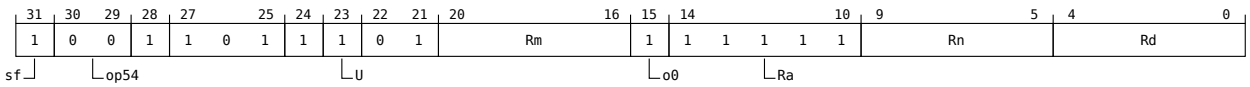
C6.2.311 UMNEGL

Unsigned multiply-negate long

This instruction multiplies two 32-bit register values, negates the product, and writes the result to the 64-bit destination register.

This is an alias of UMSUBL. This means:

- The encodings in this description are named to match the encodings of UMSUBL.
- The description of UMSUBL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding

```
UMNEGL <Xd>, <Wn>, <Wm>
```

is equivalent to

```
UMSUBL <Xd>, <Wn>, <Wm>, XZR
```

and is always the preferred disassembly.

Assembler Symbols

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Wn**> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<**Wm**> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

The description of UMSUBL gives the operational pseudocode for this instruction.

Operational Information

The description of UMSUBL gives the operational information for this instruction.

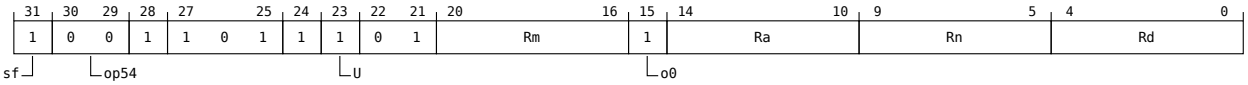
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.312 UMSUBL

Unsigned multiply-subtract long

This instruction multiplies two 32-bit register values, subtracts the product from a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias UMNEGL.



Encoding

UMSUBL <Xd>, <Wn>, <Wm>, <Xa>

Decode for this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer a = UInt(Ra);
```

Assembler Symbols

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wn**> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<**Wm**> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

<**Xa**> Is the 64-bit name of the third general-purpose source register holding the minuend, encoded in the “Ra” field.

Alias Conditions

Alias	Is preferred when
UMNEGL	Ra == '11111'

Operation

```
constant bits(32) operand1 = X[n, 32];
constant bits(32) operand2 = X[m, 32];
constant bits(64) operand3 = X[a, 64];

constant integer result = UInt(operand3) - (UInt(operand1) * UInt(operand2));

X[d, 64] = result<63:0>;
```

Operational information

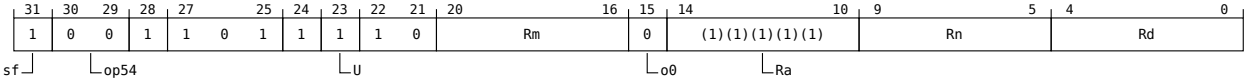
This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.313 UMULH

Unsigned multiply high

This instruction multiplies two 64-bit register values, and writes bits[127:64] of the 128-bit result to the 64-bit destination register.



Encoding

```
UMULH <Xd>, <Xn>, <Xm>
```

Decode for this encoding

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
```

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

```
constant bits(64) operand1 = X[n, 64];
constant bits(64) operand2 = X[m, 64];

constant integer result = UInt(operand1) * UInt(operand2);

X[d, 64] = result<127:64>;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

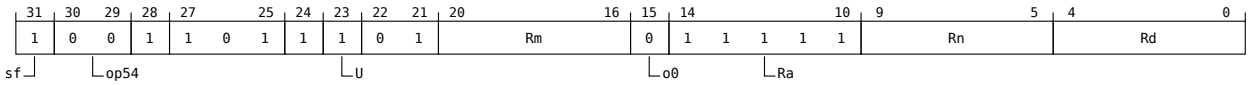
C6.2.314 UMULL

Unsigned multiply long

This instruction multiplies two 32-bit register values, and writes the result to the 64-bit destination register.

This is an alias of UMADDL. This means:

- The encodings in this description are named to match the encodings of UMADDL.
- The description of UMADDL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding

```
UMULL <Xd>, <Wn>, <Wm>
```

is equivalent to

```
UMADDL <Xd>, <Wn>, <Wm>, XZR
```

and is always the preferred disassembly.

Assembler Symbols

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wn**> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the “Rn” field.

<**Wm**> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the “Rm” field.

Operation

The description of UMADDL gives the operational pseudocode for this instruction.

Operational Information

The description of UMADDL gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

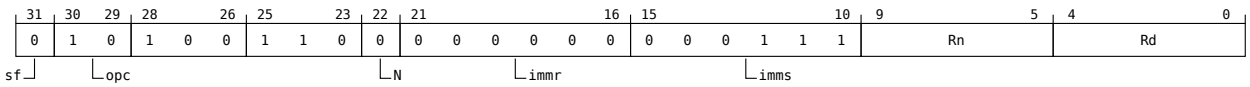
C6.2.315 UXTB

Unsigned extend byte

This instruction extracts an 8-bit value from a register, zero-extends it to the size of the register, and writes the result to the destination register.

This is an alias of UBFM. This means:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding

UXTB <Wd>, <Wn>

is equivalent to

UBFM <Wd>, <Wn>, #0, #7

and is always the preferred disassembly.

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wn**> Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

The description of UBFM gives the operational pseudocode for this instruction.

Operational Information

The description of UBFM gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

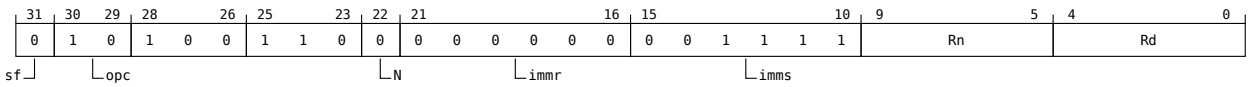
C6.2.316 UXTH

Unsigned extend halfword

This instruction extracts a 16-bit value from a register, zero-extends it to the size of the register, and writes the result to the destination register.

This is an alias of UBFM. This means:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding

UXTH <Wd>, <Wn>

is equivalent to

UBFM <Wd>, <Wn>, #0, #15

and is always the preferred disassembly.

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Wn**> Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

The description of UBFM gives the operational pseudocode for this instruction.

Operational Information

The description of UBFM gives the operational information for this instruction.

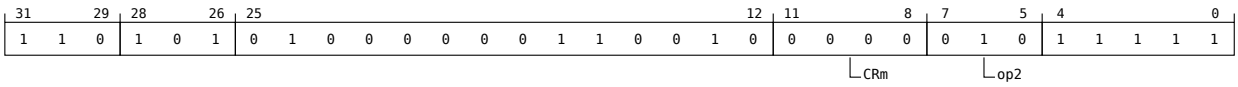
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.317 WFE

Wait for event

This instruction is a hint instruction that indicates that the PE can enter a low-power state and remain there until a wakeup event occurs. Wakeup events include the event signaled as a result of executing the `SEV` instruction on any PE in the multiprocessor system. For more information, see Wait For Event mechanism and Send event.

As described in Wait For Event mechanism and Send event, the execution of a `WFE` instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level.



Encoding

`WFE`

Decode for this encoding

```
// Empty.
```

Operation

```
Hint_WFE();
```

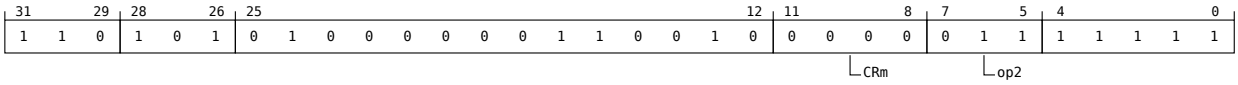

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.318 WFI

Wait for interrupt

This instruction is a hint instruction that indicates that the PE can enter a low-power state and remain there until a wakeup event occurs. For more information, see Wait For Interrupt.

As described in Wait For Interrupt, the execution of a `WFI` instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level.



Encoding

```
WFI
```

Decode for this encoding

```
// Empty.
```

Operation

```
Hint_WFI();
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.319 XPACD, XPACI, XPACLRI

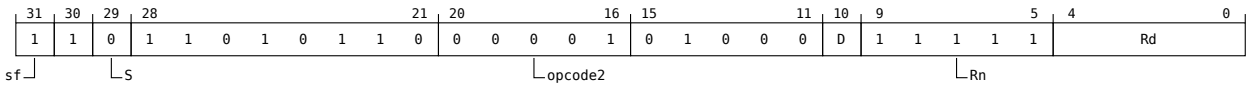
Strip Pointer Authentication Code

This instruction removes the Pointer Authentication Code from an address. The address is in the specified general-purpose register for `XPACI` and `XPACD`, and is in LR for `XPACLRI`.

The `XPACD` instruction is used for data addresses, and `XPACI` and `XPACLRI` are used for instruction addresses.

It has encodings from 2 classes: Integer and System

Integer
(FEAT_PAuth)



Encoding for the XPACD variant Applies when `(D == 1)`

```
XPACD <Xd>
```

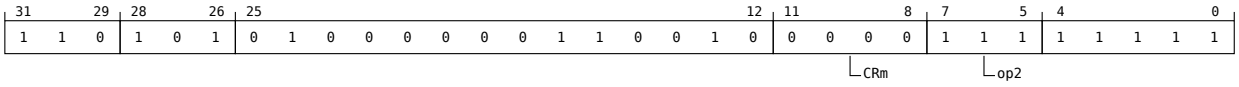
Encoding for the XPACI variant Applies when `(D == 0)`

```
XPACI <Xd>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant boolean data = (D == '1');
```

System
(FEAT_PAuth)



Encoding

```
XPACLRI
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then EndOfDecode(Decode_NOP);
constant integer d = 30;
constant boolean data = FALSE;
```

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

Operation

```
X[d, 64] = Strip(X[d, 64], data);
```

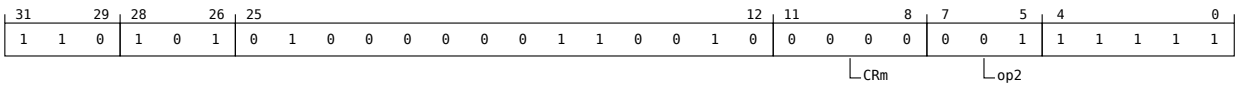

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.320 YIELD

Yield

This instruction is a hint instruction. Software with a multithreading capability can use a `YIELD` instruction to indicate to the PE that it is performing a task, for example a spin-lock, that could be swapped out to improve overall system performance. The PE can use this hint to suspend and resume multiple software threads if it supports the capability.

For more information about the recommended use of this instruction, see The YIELD instruction.



Encoding

`YIELD`

Decode for this encoding

```
// Empty.
```

Operation

```
Hint_Yield();
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter C7
A64 Advanced SIMD and Floating-point Instruction Descriptions

This chapter describes the A64 Advanced SIMD and floating-point instructions.

It contains the following sections:

- About the A64 Advanced SIMD and floating-point instructions.
- Alphabetical list of A64 Advanced SIMD and floating-point instructions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.1 About the A64 Advanced SIMD and floating-point instructions

Alphabetical list of A64 Advanced SIMD and floating-point instructions gives full descriptions of the A64 instructions that are in the following instruction groups:

- Loads and store instructions associated with the SIMD and floating-point registers.
- Data processing instructions with SIMD and floating-point registers.

A64 instruction set encoding in the A64 Instruction Encodings chapter provides an overview of the instruction encodings as part of an instruction class within a functional group.

The rest of this section is a general description of the SIMD and floating-point instructions. It contains the following subsections:

- Register size.
- Output element control.
- Data types.
- Condition flags and related instructions.

C7.1.1 Register size

A64 provides a comprehensive set of packed Single Instruction Multiple Data (SIMD) and scalar operations using data held in the 32 entry 128-bit wide SIMD and floating-point register file.

Each SIMD and floating-point register can be used to hold:

- A single scalar value of the floating-point or integer type.
- A 64-bit wide vector containing one or more elements.
- A 128-bit wide vector containing two or more elements.

Where the entire 128-bit wide register is not fully utilized, the vector or scalar quantity is held in the least significant bits of the register, with the most significant bits being cleared to zero on a write, see Advanced SIMD vector formats.

The following instructions can insert data into individual elements within a SIMD and floating-pointer register without clearing the remaining bits to zero:

- Insert vector element from another vector element or general-purpose register, `INS`.
- Load structure into a single lane, for example `LD3`.
- All second-part narrowing operations, for example `SHRN2`.

C7.1.2 Output element control

When FEAT_AFP is implemented, the FPCR.NEP bit controls how output elements are determined for the scalar Advanced SIMD instructions for elements other than the lowest element of the vector.

If FPCR.NEP == 1, the following instructions determine output elements as follows:

- The 3-input floating-point scalar versions of FMLA (by element) and FMLS (by element) take output elements other than the lowest element from the <Hd>, <Sd>, or <Dd> register.
- The 3-input floating-point FMADD, FMSUB, FNMADD, and FNMSUB instructions take output elements other than the lowest element from the <Ha>, <Sa>, or <Da> register.
- The 2-input floating-point scalar versions of FCMGE (register), FCMGT (register), FCMEQ (register), FACGE, FACGT, take output elements other than the lowest element from the <Hm>, <Sm>, or <Dm> register.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The 2-input floating-point scalar versions of FMULX, FRECPS, FRSQRTS, FABD, FMUL (by element), FMUL (scalar), FDIV (scalar), FADD (scalar), FSUB (scalar), FMAX (scalar), FMIN (scalar), FMAXNM (scalar), FMINNM (scalar), FNMUL (scalar), take output elements other than the lowest element from the <Hn>, <Sn>, or <Dn> register.
- For 1-input floating-point scalar versions of the instructions FCVTNS (vector), FCVTMS (vector), FCVTAS (vector), FCVTPS (vector), SCVTF (vector, integer), UCVTF (vector, integer) FCVTZS (vector, integer), FCVTZU (vector, integer), FCVTNU (vector), FCVTMU (vector), FCVTAU (vector), FCVTPU (vector), SCVTF (vector, fixed-point), UCVTF (vector, fixed-point), FCVTZS (vector, fixed-point), FCVTZU (vector, fixed-point), SCVTF (scalar, integer), UCVTF (scalar, integer), SCVTF (scalar, fixed-point), UCVTF (scalar, fixed-point), FCVT, FCVTXN, FRECPE, FRECPX, FRSQRTE, FABS (scalar), FNEG (scalar), FSQRT (scalar), FRINTN (scalar), FRINTP (scalar), FRINTM (scalar), FRINTZ (scalar), FRINTA (scalar), FRINTI (scalar), FRINTX (scalar), take output elements other than the lowest element from the <Hd>, <Sd>, or <Dd> register.

C7.1.3 Data types

The A64 instruction set provides support for arithmetic, conversion, and bitwise operations on:

- Half-precision, single-precision, and double-precision floating-points.
- Signed and unsigned integers.
- Polynomials over {0, 1}.
- When FEAT_FCMA is implemented, complex numbers.

For all AArch64 floating-point operations, including SIMD operations, the rounding mode and exception trap handling are controlled by the FPCR.

Note

In AArch64 state, floating-point multiply-add operations are always performed as fused operations.

In addition to operations that consume and produce values of the same width and type, the A64 instruction set supports SIMD and scalar operations that produce a wider or narrower vector result:

- Where a SIMD operation narrows a 128-bit vector to a 64-bit vector, the A64 instruction set provides a second-part operation, for example SHRN2, that can pack the result of a second operation into the upper part of the same destination register.
- Where a SIMD operation widens a 64-bit vector to a 128-bit vector, the A64 instruction set provides a second-part operation, for example SMLAL2, that can extract the source from the upper 64 bits of the source registers.

All SIMD operations that could produce side-effects that are not limited to the destination SIMD and floating-point register, for example a potential update of FPSR.Q or FPSR.IDC, have a dedicated scalar variant to support the use of SIMD with loops requiring specialized head or tail handling, or both.

C7.1.4 Condition flags and related instructions

The A64 instruction set provides support for flag setting and conditional operations on the SIMD and floating-point register file:

- Floating-point FCSEL and FCCMP instructions are equivalent to the integer CSEL and CCMP instructions.
- Floating-point FCMP, FCMPE, FCCMP, and FCCMP instructions set the PSTATE.{N, Z, C, V} flags based on the result of the floating-point comparison.
- Floating-point FJCVTZS instruction sets the PSTATE.Z flag if the result of the conversion, when converted back to a double-precision floating-point number, gives precisely the same value as the original. Other PSTATE flags are cleared by this instruction.
- Floating-point and integer instructions provide a means of producing either a scalar or a vector mask based on a comparison in a SIMD and floating-point register, for example FCMEQ.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

A64 instructions store the result of an `FCMP` or `FCMPE` operation in the PSTATE.{N, Z, C, V} field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2 Alphabetical list of A64 Advanced SIMD and floating-point instructions

This section lists every section in the Advanced SIMD and floating-point categories of the A64 instruction set. For details of the format used, see Structure of the A64 assembler language.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.1 ABS

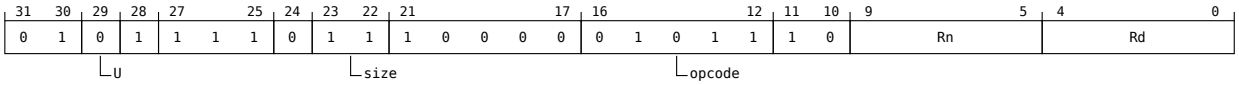
Absolute value (vector)

This instruction calculates the absolute value of each vector element in the source SIMD&FP register, puts the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



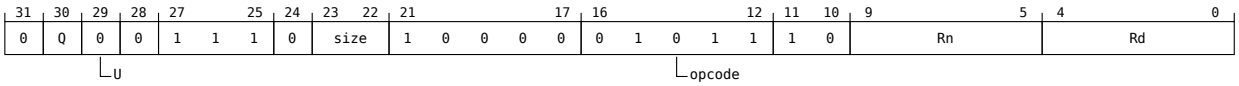
Encoding

```
ABS D<d>, D<n>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
ABS <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<d> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**n**> Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    element = Abs(element);
    Elem[result, e, esize] = element<esize-1:0>;

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.2 ADD (vector)

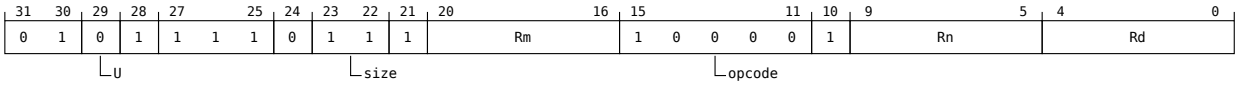
Add (vector)

This instruction adds corresponding elements in the two source SIMD&FP registers, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



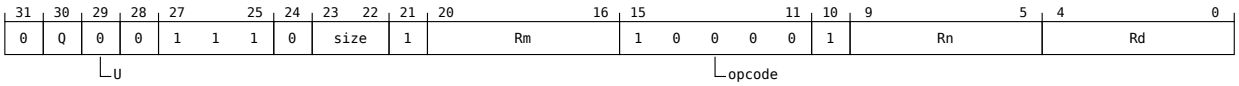
Encoding

```
ADD D<d>, D<n>, D<m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
ADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**d**> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**n**> Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<**m**> Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = element1 + element2;
V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.3 ADDHN, ADDHN2

Add returning high narrow

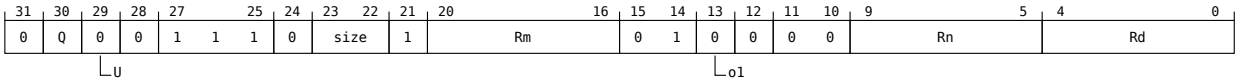
This instruction adds each vector element in the first source SIMD&FP register to the corresponding vector element in the second source SIMD&FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register.

The results are truncated. For rounded results, see RADDHN.

The ADDHN instruction writes the vector to the lower half of the destination register and clears the upper half. The ADDHN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers, not all the same type
(FEAT_AdvSIMD)



Encoding

```
ADDHN{2} <Vd>.<Tb>, <Vn>.<Ta>, <Vm>.<Ta>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;
constant boolean round = FALSE;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Tb**> Is an arrangement specifier, encoded in “size:Q”:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Ta**> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(2*datasize) operand1 = V[n, 2*datasize];
constant bits(2*datasize) operand2 = V[m, 2*datasize];
bits(datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, 2*esize]);
    element2 = UInt(Elem[operand2, e, 2*esize]);
    sum = element1 + element2;
    sum = RShr(sum, esize, round);
    Elem[result, e, esize] = sum<esize-1:0>;

Vpart[d, part, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

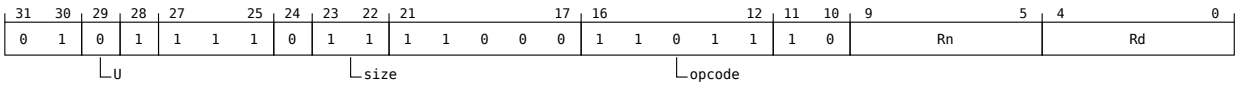
C7.2.4 ADDP (scalar)

Add pair of elements (scalar)

This instruction adds two vector elements in the source SIMD&FP register and writes the scalar result into the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Advanced SIMD
(FEAT_AdvSIMD)



Encoding

```
ADDP D<d>, <Vn>.2D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 64;
constant integer datasize = 128;
```

Assembler Symbols

<d> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
V[d, esize] = IntReduce(ReduceOp_ADD, operand, esize);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

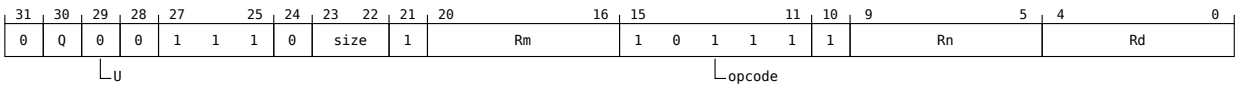
C7.2.5 ADDP (vector)

Add pairwise (vector)

This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements from the concatenated vector, adds each pair of values together, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type
(FEAT_AdvSIMD)



Encoding

```
ADDP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
constant bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[concat, 2*e, esize];
    element2 = Elem[concat, (2*e)+1, esize];
    Elem[result, e, esize] = element1 + element2;

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

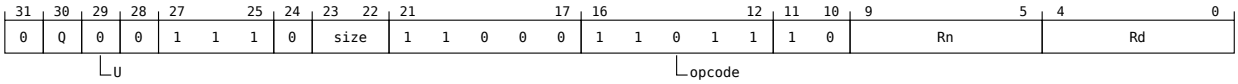
C7.2.6 ADDV

Add across vector

This instruction adds every vector element in the source SIMD&FP register together, and writes the scalar result to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Advanced SIMD
(FEAT_AdvSIMD)



Encoding

```
ADDV <V><d>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '100' then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
```

Assembler Symbols

<**V**> Is the destination width specifier, encoded in "size":

size	<V>
00	B
01	H
10	S
11	RESERVED

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<**T**> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	Q	<T>
01	1	8H
10	0	RESERVED
10	1	4S
11	x	RESERVED

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
V[d, esize] = IntReduce(ReduceOp_ADD, operand, esize);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

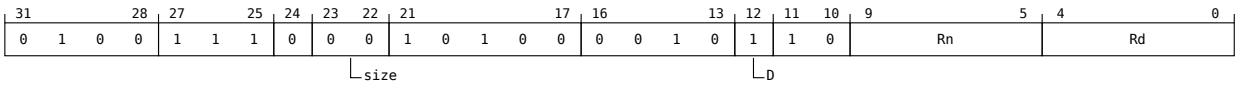
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.7 AESD

AES single round decryption

AES single round decryption.

Advanced SIMD
(FEAT_AES)



Encoding

```
AESD <Vd>.16B, <Vn>.16B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AES) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.

<**Vn**> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(128) operand1 = V[d, 128];
constant bits(128) operand2 = V[n, 128];
bits(128) result = operand1 EOR operand2;
result = AESInvShiftRows(result);
V[d, 128] = AESInvSubBytes(result);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

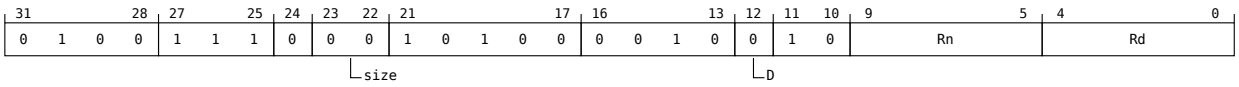
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.8 AESE

AES single round encryption

AES single round encryption.

Advanced SIMD
(FEAT_AES)



Encoding

```
AESE <Vd>.16B, <Vn>.16B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AES) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.

<**Vn**> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(128) operand1 = V[d, 128];
constant bits(128) operand2 = V[n, 128];
bits(128) result = operand1 EOR operand2;
result = AESShiftRows(result);
V[d, 128] = AESSubBytes(result);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

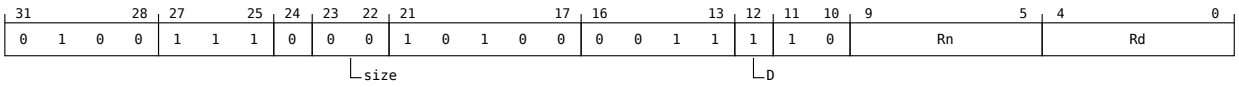
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.9 AESIMC

AES inverse mix columns

AES inverse mix columns.

Advanced SIMD
(FEAT_AES)



Encoding

```
AESIMC <Vd>.16B, <Vn>.16B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AES) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(128) operand = V[n, 128];
V[d, 128] = AESInvMixColumns(operand);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

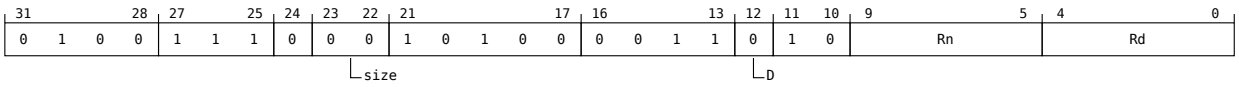
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.10 AESMC

AES mix columns

AES mix columns.

Advanced SIMD
(FEAT_AES)



Encoding

```
AESMC <Vd>.16B, <Vn>.16B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AES) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(128) operand = V[n, 128];
V[d, 128] = AESMixColumns(operand);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

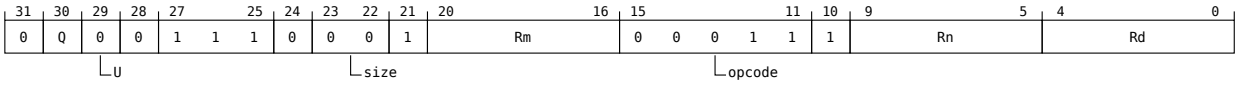
C7.2.11 AND (vector)

Bitwise AND (vector)

This instruction performs a bitwise AND between the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type
(FEAT_AdvSIMD)



Encoding

```
AND <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 64 << UInt(Q);
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> Is an arrangement specifier, encoded in "Q":

Q	<T>
0	8B
1	16B

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
V[d, datasize] = operand1 AND operand2;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

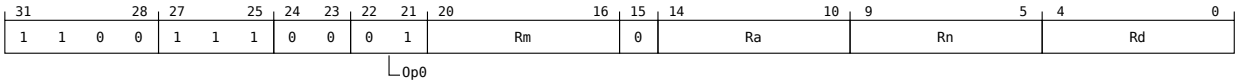
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.12 BCAX

Bit clear and exclusive-OR

This instruction performs a bitwise AND of the 128-bit vector in a source SIMD&FP register and the complement of the vector in another source SIMD&FP register, then performs a bitwise exclusive-OR of the resulting vector and the vector in a third source SIMD&FP register, and writes the result to the destination SIMD&FP register.

Advanced SIMD
(FEAT_SHA3)



Encoding

```
BCAX <Vd>.16B, <Vn>.16B, <Vm>.16B, <Va>.16B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA3) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer a = UInt(Ra);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

<Va> Is the name of the third SIMD&FP source register, encoded in the “Ra” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(128) operand1 = V[m, 128];
constant bits(128) operand2 = V[n, 128];
constant bits(128) operand3 = V[a, 128];

V[d, 128] = operand2 EOR (operand1 AND NOT(operand3));
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

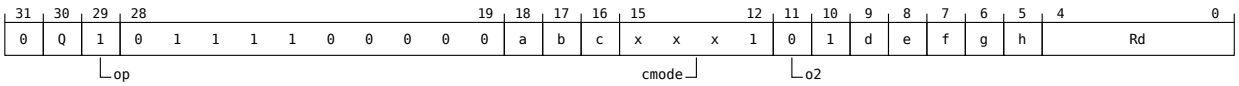
C7.2.13 BIC (vector, immediate)

Bitwise bit clear (vector, immediate)

This instruction reads each vector element from the destination SIMD&FP register, performs a bitwise AND between each result and the complement of an immediate constant, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Shifted immediate
(FEAT_AdvSIMD)



Encoding for the 16-bit variant Applies when `(cmode == 10x1)`

```
BIC <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

Encoding for the 32-bit variant Applies when `(cmode == 0xx1)`

```
BIC <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer rd = UInt(Rd);
constant integer datasize = 64 << UInt(Q);
constant bits(64) imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
constant bits(datasize) imm = Replicate(imm64, datasize DIV 64);
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP register, encoded in the "Rd" field.

<**T**> For the "16-bit" variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the "32-bit" variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	2S
1	4S

<**imm8**> Is an 8-bit immediate encoded in "a:b:c:d:e:f:g:h".

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**amount**> For the “16-bit” variant: is the shift amount encoded in “cmode<1>”:

cmode<1>	<**amount**>
0	0
1	8

defaulting to 0 if LSL is omitted.

For the “32-bit” variant: is the shift amount encoded in “cmode<2:1>”:

cmode<2:1>	<**amount**>
00	0
01	8
10	16
11	24

defaulting to 0 if LSL is omitted.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[rd, datasize];
V[rd, datasize] = operand AND NOT(imm);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

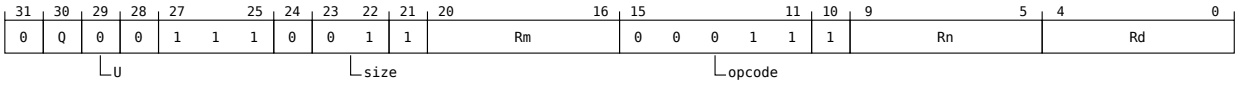
C7.2.14 BIC (vector, register)

Bitwise bit clear (vector, register)

This instruction performs a bitwise AND between the first source SIMD&FP register and the complement of the second source SIMD&FP register, and writes the result to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type
(FEAT_AdvSIMD)



Encoding

```
BIC <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 64 << UInt(Q);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "Q":

Q	<T>
0	8B
1	16B

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = NOT(V[m, datasize]);
V[d, datasize] = operand1 AND operand2;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

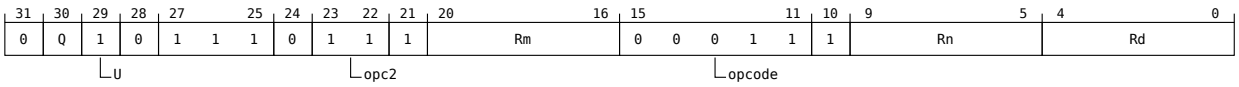
C7.2.15 BIF

Bitwise insert if false

This instruction inserts each bit from the first source SIMD&FP register into the destination SIMD&FP register if the corresponding bit of the second source SIMD&FP register is 0, otherwise leaves the bit in the destination register unchanged.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type
(FEAT_AdvSIMD)



Encoding

```
BIF <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 64 << UInt(Q);
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	8B
1	16B

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(datasize) operand1 = V[d, datasize];
constant bits(datasize) operand2 = NOT(V[m, datasize]);
constant bits(datasize) operand3 = V[n, datasize];
V[d, datasize] = operand1 EOR ((operand1 EOR operand3) AND operand2);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

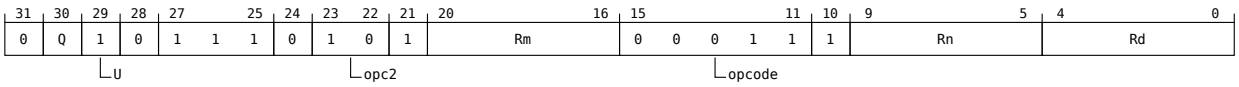
C7.2.16 BIT

Bitwise insert if true

This instruction inserts each bit from the first source SIMD&FP register into the SIMD&FP destination register if the corresponding bit of the second source SIMD&FP register is 1, otherwise leaves the bit in the destination register unchanged.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type
(FEAT_AdvSIMD)



Encoding

```
BIT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 64 << UInt(Q);
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	8B
1	16B

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(datasize) operand1 = V[d, datasize];
constant bits(datasize) operand2 = V[m, datasize];
constant bits(datasize) operand3 = V[n, datasize];
V[d, datasize] = operand1 EOR ((operand1 EOR operand3) AND operand2);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

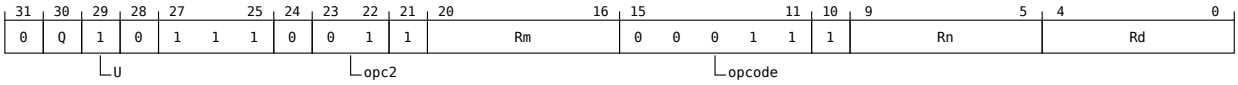
C7.2.17 BSL

Bitwise select

This instruction sets each bit in the destination SIMD&FP register to the corresponding bit from the first source SIMD&FP register when the original destination bit was 1, otherwise from the second source SIMD&FP register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type
(FEAT_AdvSIMD)



Encoding

```
BSL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 64 << UInt(Q);
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	8B
1	16B

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(datasize) operand1 = V[m, datasize];
constant bits(datasize) operand2 = V[d, datasize];
constant bits(datasize) operand3 = V[n, datasize];
V[d, datasize] = operand1 EOR ((operand1 EOR operand3) AND operand2);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

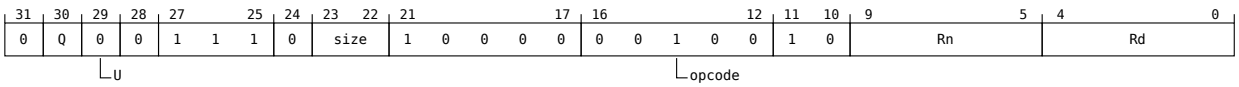
C7.2.18 CLS (vector)

Count leading sign bits (vector)

This instruction determines the number of consecutive bits following the most significant bit that are the same as the most significant bit in each vector element in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. The result does not include the most significant bit itself.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector
(FEAT_AdvSIMD)



Encoding

```
CLS <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(datasize) result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
for e = 0 to elements-1
    constant integer count = CountLeadingSignBits(Elem[operand, e, esize]);
    Elem[result, e, esize] = count<esize-1:0>;
V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

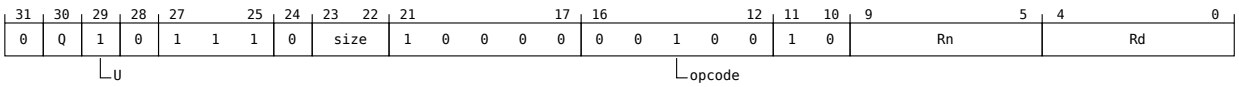
C7.2.19 CLZ (vector)

Count leading zero bits (vector)

This instruction determines the number of consecutive zeros, starting from the most significant bit, in each vector element in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector
(FEAT_AdvSIMD)



Encoding

```
CLZ <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(datasize) result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
for e = 0 to elements-1
    constant integer count = CountLeadingZeroBits(Elem[operand, e, esize]);
    Elem[result, e, esize] = count<esize-1:0>;
V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.20 CMEQ (register)

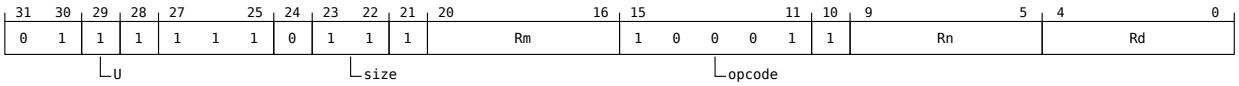
Compare bitwise equal (vector)

This instruction compares each vector element from the first source SIMD&FP register with the corresponding vector element from the second source SIMD&FP register, and if the comparison is equal sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



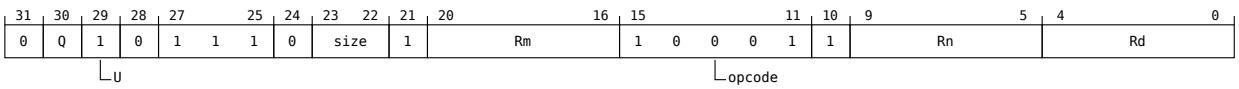
Encoding

```
CMEQ D<d>, D<n>, D<m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
CMEQ <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**d**> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<**n**> Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<**m**> Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = if element1 == element2 then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.21 CMEQ (zero)

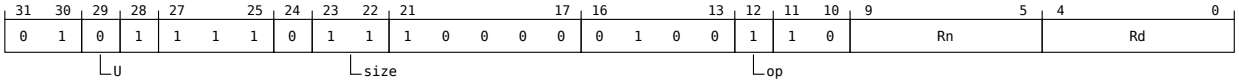
Compare bitwise equal to zero (vector)

This instruction reads each vector element in the source SIMD&FP register and if the value is equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



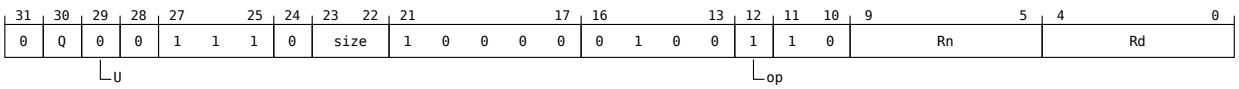
Encoding

```
CMEQ D<d>, D<n>, #0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size != '11' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
CMEQ <Vd>.<T>, <Vn>.<T>, #0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**n**> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    Elem[result, e, esize] = if element == 0 then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.22 CMGE (register)

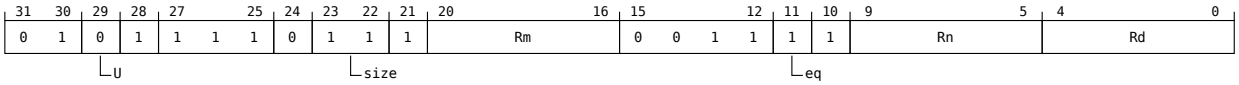
Compare signed greater than or equal (vector)

This instruction compares each vector element in the first source SIMD&FP register with the corresponding vector element in the second source SIMD&FP register and if the first signed integer value is greater than or equal to the second signed integer value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



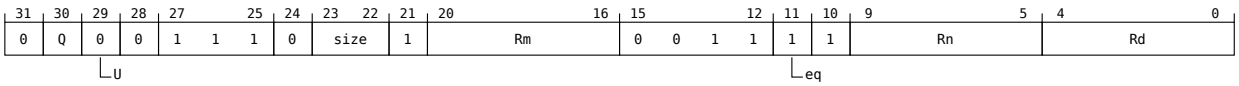
Encoding

```
CMGE D<d>, D<n>, D<m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size != '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
CMGE <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<**d**> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<**n**> Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<**m**> Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    Elem[result, e, esize] = if element1 >= element2 then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.23 CMGE (zero)

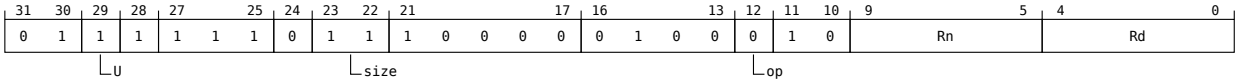
Compare signed greater than or equal to zero (vector)

This instruction reads each vector element in the source SIMD&FP register and if the signed integer value is greater than or equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



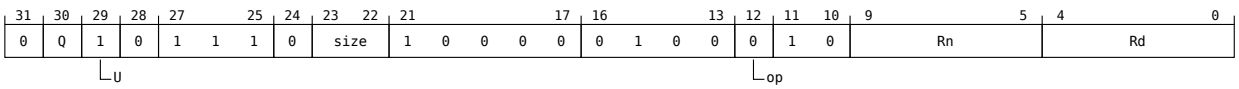
Encoding

```
CMGE D<d>, D<n>, #0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size != '11' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
CMGE <Vd>.<T>, <Vn>.<T>, #0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**n**> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    Elem[result, e, esize] = if element >= 0 then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.24 CMGT (register)

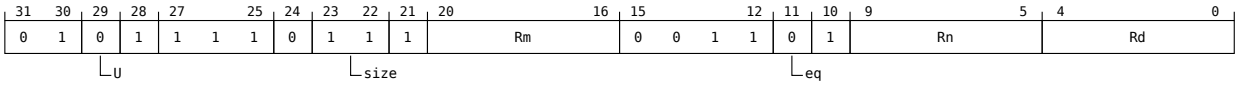
Compare signed greater than (vector)

This instruction compares each vector element in the first source SIMD&FP register with the corresponding vector element in the second source SIMD&FP register and if the first signed integer value is greater than the second signed integer value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



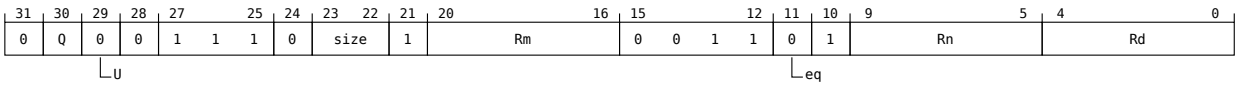
Encoding

```
CMGT D<d>, D<n>, D<m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size != '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
CMGT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<**d**> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<**n**> Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<**m**> Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    Elem[result, e, esize] = if element1 > element2 then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.25 CMGT (zero)

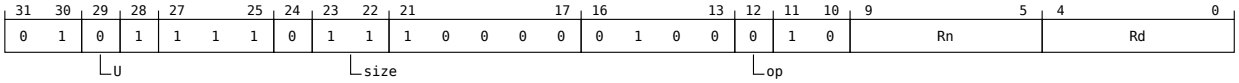
Compare signed greater than zero (vector)

This instruction reads each vector element in the source SIMD&FP register and if the signed integer value is greater than zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



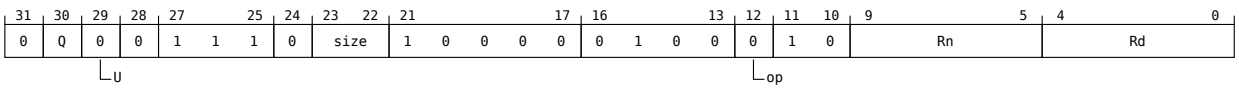
Encoding

```
CMGT D<d>, D<n>, #0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size != '11' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
CMGT <Vd>.<T>, <Vn>.<T>, #0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**d**> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<**n**> Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    Elem[result, e, esize] = if element > 0 then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.26 CMHI (register)

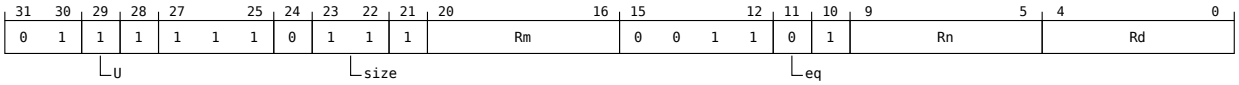
Compare unsigned higher (vector)

This instruction compares each vector element in the first source SIMD&FP register with the corresponding vector element in the second source SIMD&FP register and if the first unsigned integer value is greater than the second unsigned integer value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



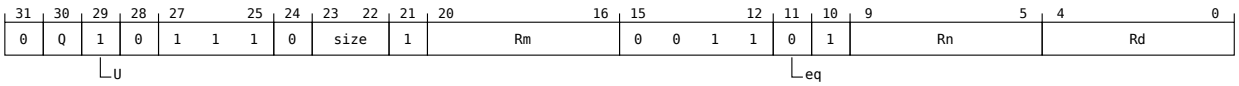
Encoding

```
CMHI D<d>, D<n>, D<m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size != '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
CMHI <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<**d**> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<**n**> Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<**m**> Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;

for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    element2 = UInt(Elem[operand2, e, esize]);
    Elem[result, e, esize] = if element1 > element2 then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.27 CMHS (register)

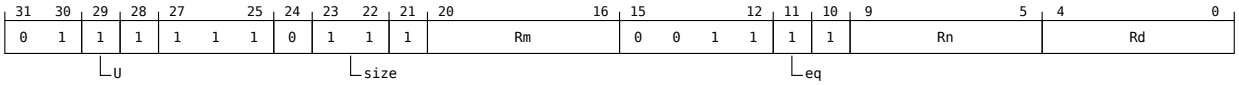
Compare unsigned higher or same (vector)

This instruction compares each vector element in the first source SIMD&FP register with the corresponding vector element in the second source SIMD&FP register and if the first unsigned integer value is greater than or equal to the second unsigned integer value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



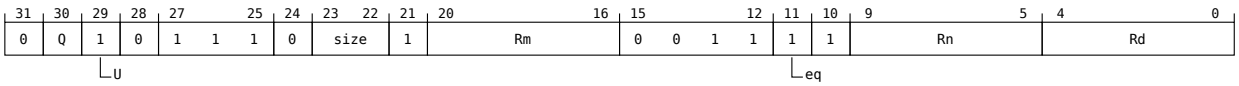
Encoding

```
CMHS D<d>, D<n>, D<m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size != '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
CMHS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<**d**> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<**n**> Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<**m**> Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;

for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    element2 = UInt(Elem[operand2, e, esize]);
    Elem[result, e, esize] = if element1 >= element2 then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.28 CMLE (zero)

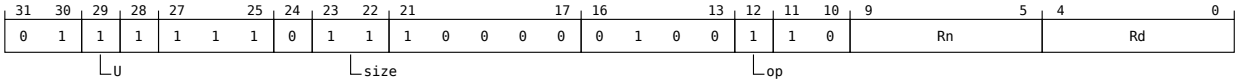
Compare signed less than or equal to zero (vector)

This instruction reads each vector element in the source SIMD&FP register and if the signed integer value is less than or equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



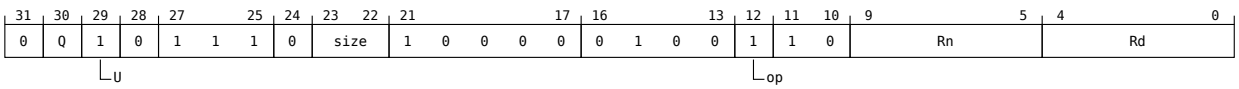
Encoding

```
CMLE D<d>, D<n>, #0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size != '11' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
CMLE <Vd>.<T>, <Vn>.<T>, #0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**n**> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    Elem[result, e, esize] = if element <= 0 then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.29 CMLT (zero)

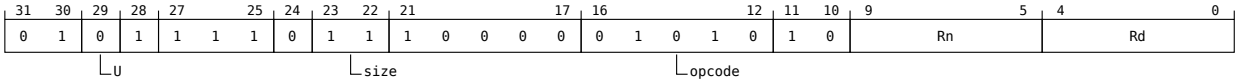
Compare signed less than zero (vector)

This instruction reads each vector element in the source SIMD&FP register and if the signed integer value is less than zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



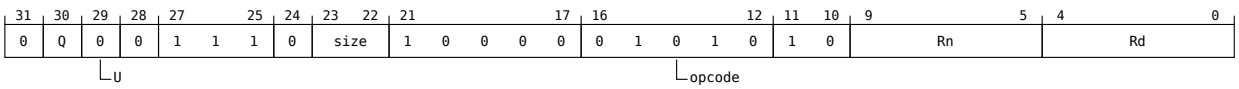
Encoding

```
CMLT D<d>, D<n>, #0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size != '11' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
CMLT <Vd>.<T>, <Vn>.<T>, #0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**n**> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    Elem[result, e, esize] = if element < 0 then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.30 CMTST

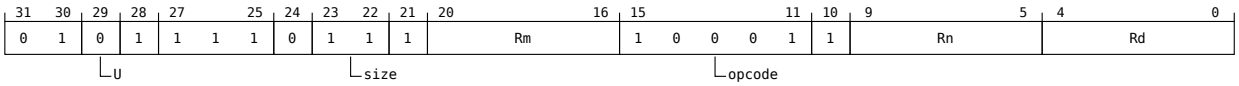
Compare bitwise test bits nonzero (vector)

This instruction reads each vector element in the first source SIMD&FP register, performs an AND with the corresponding vector element in the second source SIMD&FP register, and if the result is not zero, sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



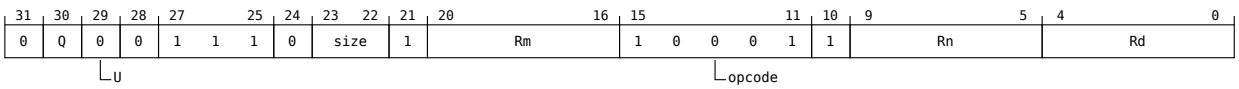
Encoding

```
CMTST D<d>, D<n>, D<m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
CMTST <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**d**> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<**n**> Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<**m**> Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = if !IsZero(element1 AND element2) then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

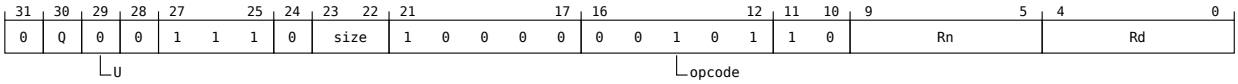
C7.2.31 CNT

Population count per byte

This instruction determines the number of bits that have a value of one in each vector element in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector
(FEAT_AdvSIMD)



Encoding

```
CNT <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size != '00' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 8;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV 8;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	x	RESERVED
1x	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(datasize) result;

for e = 0 to elements-1
    constant integer count = BitCount(Elem[operand, e, esize]);
    Elem[result, e, esize] = count<esize-1:0>;
V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.32 DUP (element)

Duplicate vector element to vector or scalar

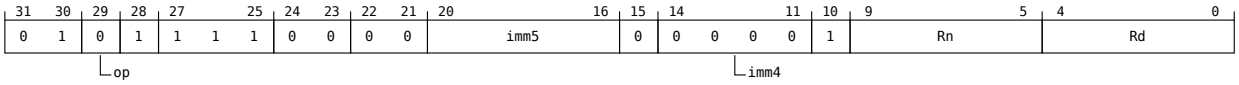
This instruction duplicates the vector element at the specified element index in the source SIMD&FP register into a scalar or each element in a vector, and writes the result to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias MOV (scalar).

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



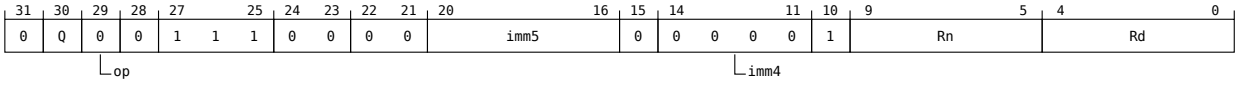
Encoding

```
DUP <V><d>, <Vn>.<T>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if imm5 == 'x0000' then EndOfDecode(Decode_UNDEF);
constant integer size = LowestSetBitNZ(imm5<3:0>);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer index = UInt(imm5<4:size+1>);
constant integer idxdsize = 64 << UInt(imm5<4>);
constant integer esize = 8 << size;
constant integer datasize = esize;
constant integer elements = 1;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
DUP <Vd>.<T>, <Vn>.<Ts>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if imm5 == 'x0000' then EndOfDecode(Decode_UNDEF);
if imm5 == 'x1000' && Q == '0' then EndOfDecode(Decode_UNDEF);
constant integer size = LowestSetBitNZ(imm5<3:0>);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer index = UInt(imm5<4:size+1>);
constant integer idxdsize = 64 << UInt(imm5<4>);
constant integer esize = 8 << size;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**V**> Is the destination width specifier, encoded in "imm5":

imm5	<V>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<**T**> For the "Scalar" variant: is the element width specifier, encoded in "imm5":

imm5	<T>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

For the "Vector" variant: is an arrangement specifier, encoded in "imm5:Q":

imm5	Q	<T>
x0000	x	RESERVED
xxxx1	0	8B
xxxx1	1	16B
xxx10	0	4H
xxx10	1	8H
xx100	0	2S
xx100	1	4S
x1000	0	RESERVED
x1000	1	2D

<**index**> Is the element index encoded in "imm5":

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

imm5	<index>
x0000	RESERVED
xxxx1	UInt(imm5<4:1>)
xxx10	UInt(imm5<4:2>)
xx100	UInt(imm5<4:3>)
x1000	UInt(imm5<4>)

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Ts**> Is an element size specifier, encoded in “imm5”:

imm5	<Ts>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(idxdsize) operand = V[n, idxdsize];
bits(datasize) result;
bits(esize) element;

element = Elem[operand, index, esize];
for e = 0 to elements-1
    Elem[result, e, esize] = element;
V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

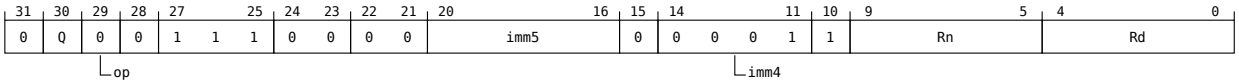
C7.2.33 DUP (general)

Duplicate general-purpose register to vector

This instruction duplicates the contents of the source general-purpose register into a scalar or each element in a vector, and writes the result to the SIMD&FP destination register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Advanced SIMD
(FEAT_AdvSIMD)



Encoding

```
DUP <Vd>.<T>, <R><n>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if imm5 == 'x0000' then EndOfDecode(Decode_UNDEF);
if imm5 == 'x1000' && Q == '0' then EndOfDecode(Decode_UNDEF);
constant integer size = LowestSetBitNZ(imm5<3:0>);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
// imm5<4:size+1> is IGNORED
constant integer esize = 8 << size;
constant integer datasize = 64 << UInt(Q);
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> Is an arrangement specifier, encoded in "imm5:Q":

imm5	Q	<T>
x0000	x	RESERVED
xxxx1	0	8B
xxxx1	1	16B
xxx10	0	4H
xxx10	1	8H
xx100	0	2S
xx100	1	4S
x1000	0	RESERVED
x1000	1	2D

<**R**> Is the width specifier for the general-purpose source register, encoded in "imm5":

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

imm5	<R>
x0000	RESERVED
xxxx1	W
xxx10	W
xx100	W
x1000	X

Unspecified bits in “imm5” are ignored but should be set to zero by an assembler.

<**n**> Is the number [0-30] of the general-purpose source register or ZR (31), encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(esize) element = X[n, esize];
constant integer elements = datasize DIV esize;
bits(datasize) result;

for e = 0 to elements-1
    Elem[result, e, esize] = element;
V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

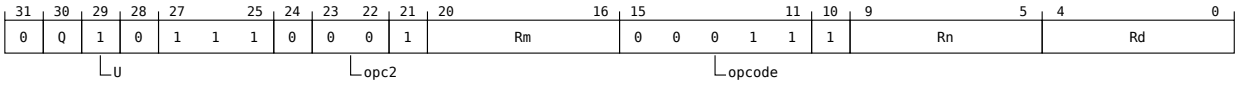
C7.2.34 EOR (vector)

Bitwise exclusive-OR (vector)

This instruction performs a bitwise exclusive-OR operation between the two source SIMD&FP registers, and places the result in the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type
(FEAT_AdvSIMD)



Encoding

```
EOR <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 64 << UInt(Q);
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	8B
1	16B

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(datasize) operand1 = V[m, datasize];
constant bits(datasize) operand2 = Zeros(datasize);
constant bits(datasize) operand3 = Ones(datasize);
constant bits(datasize) operand4 = V[n, datasize];
V[d, datasize] = operand1 EOR ((operand2 EOR operand4) AND operand3);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

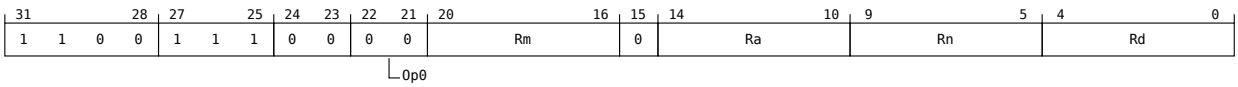
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.35 EOR3

Three-way exclusive-OR

This instruction performs a three-way exclusive-OR of the values in the three source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

Advanced SIMD
(FEAT_SHA3)



Encoding

```
EOR3 <Vd>.16B, <Vn>.16B, <Vm>.16B, <Va>.16B
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA3) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer a = UInt(Ra);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

<Va> Is the name of the third SIMD&FP source register, encoded in the “Ra” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(128) operand1 = V[m, 128];
constant bits(128) operand2 = V[n, 128];
constant bits(128) operand3 = V[a, 128];
V[d, 128] = operand2 EOR operand1 EOR operand3;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

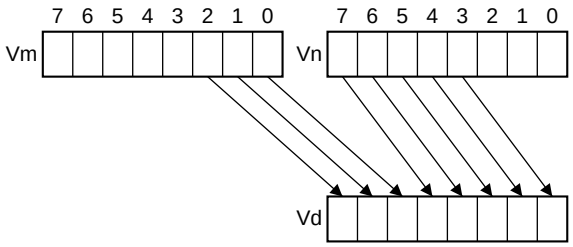
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.36 EXT

Extract vector from pair of vectors

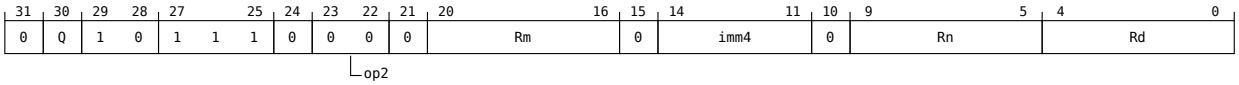
This instruction extracts the lowest vector elements from the second source SIMD&FP register and the highest vector elements from the first source SIMD&FP register, concatenates the results into a vector, and writes the vector to the destination SIMD&FP register vector. The index value specifies the lowest vector element to extract from the first source register, and consecutive elements are extracted from the first, then second, source registers until the destination vector is filled.

The following figure shows an example of the operation of EXT doubleword operation for Q = 0 and imm4<2:0> = 3.



Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Advanced SIMD
(FEAT_AdvSIMD)



Encoding

```
EXT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>, #<index>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if Q == '0' && imm4<3> == '1' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 64 << UInt(Q);
constant integer position = 8 * UInt(imm4);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "Q":

Q	<T>
0	8B
1	16B

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

<**index**> Is the lowest numbered byte element to be extracted, encoded in “Q:imm4”:

Q	imm4<3>	<index>
0	0	UInt(imm4<2:0>)
0	1	RESERVED
1	x	UInt(imm4)

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) hi = V[m, datasize];
constant bits(datasize) lo = V[n, datasize];
constant bits(datasize*2) concat = hi : lo;

V[d, datasize] = concat<(position+datasize)-1:position>;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.37 FABD

Floating-point absolute difference (vector)

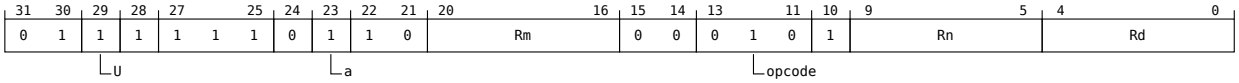
This instruction subtracts the floating-point values in the elements of the second source SIMD&FP register, from the corresponding floating-point values in the elements of the first source SIMD&FP register, places the absolute value of each result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision , Scalar single-precision and double-precision , Vector half-precision and Vector single-precision and double-precision

Scalar half-precision
(FEAT_AdvSIMD && FEAT_FP16)



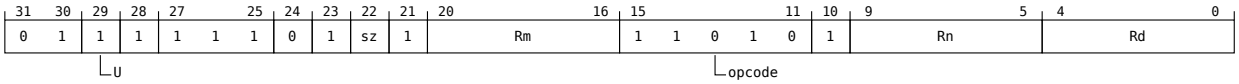
Encoding

```
FABD <Hd>, <Hn>, <Hm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 16;
constant integer datasize = esize;
constant integer elements = 1;
```

Scalar single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

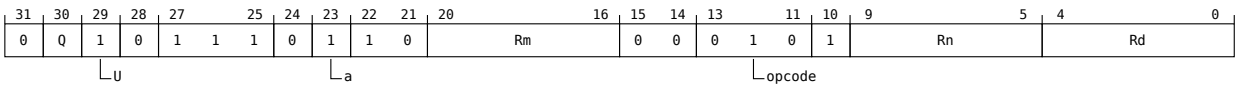
```
FABD <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 32 << UInt(sz);
constant integer datasize = esize;
constant integer elements = 1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Vector half-precision
(FEAT_AdvSIMD && FEAT_FP16)



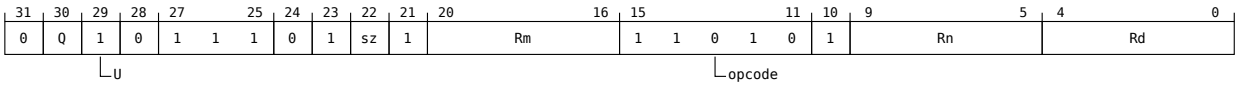
Encoding

```
FABD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Vector single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

```
FABD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Hm**> Is the 16-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<**V**> Is a width specifier, encoded in “sz”:

sz	<**V**>
0	S

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	<V>
1	D

<**d**> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<**n**> Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<**m**> Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> For the “Vector half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Vector single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];

bits(esize) element1;
bits(esize) element2;
bits(datasize) result;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    constant bits(esize) diff = FPSub(element1, element2, FPCR);
    Elem[result, e, esize] = FPAbs(diff, FPCR);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.38 FABS (vector)

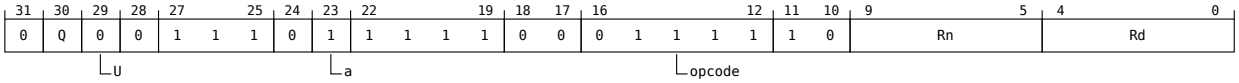
Floating-point absolute value (vector)

This instruction calculates the absolute value of each vector element in the source SIMD&FP register, writes the result to a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(FEAT_AdvSIMD && FEAT_FP16)



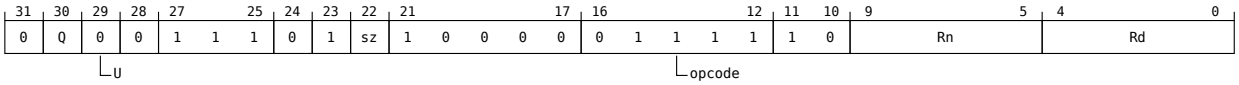
Encoding

```
FABS <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

```
FABS <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**T**> For the “Half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(datasize) result;

for e = 0 to elements-1
    constant bits(esize) element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPAbs(element, FPCR);

V[d, datasize] = result;
```

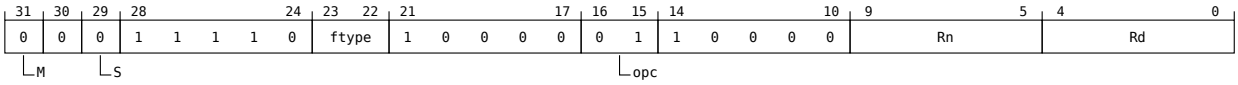

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.39 FABS (scalar)

Floating-point absolute value (scalar)

This instruction calculates the absolute value in the SIMD&FP source register and writes the result to the SIMD&FP destination register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Encoding for the Half-precision variant
(FEAT_FP16) Applies when `(ftype == 11)`

```
FABS <Hd>, <Hn>
```

Encoding for the Single-precision variant
(FEAT_FP) Applies when `(ftype == 00)`

```
FABS <Sd>, <Sn>
```

Encoding for the Double-precision variant
(FEAT_FP) Applies when `(ftype == 01)`

```
FABS <Dd>, <Dn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if ftype == '10' then EndOfDecode(Decode_UNDEF);
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 8 << UInt(ftype EOR '10');
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<**Sd**> Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Sn**> Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<**Dd**> Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Dn**> Is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(esize) operand = V[n, esize];

constant bits(esize) result = FPAbs(operand, FPCR);

V[d, esize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.40 FACGE

Floating-point absolute compare greater than or equal (vector)

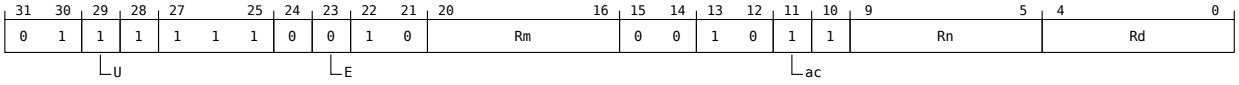
This instruction compares the absolute value of each floating-point value in the first source SIMD&FP register with the absolute value of the corresponding floating-point value in the second source SIMD&FP register and if the first value is greater than or equal to the second value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision , Scalar single-precision and double-precision , Vector half-precision and Vector single-precision and double-precision

Scalar half-precision
(FEAT_AdvSIMD && FEAT_FP16)



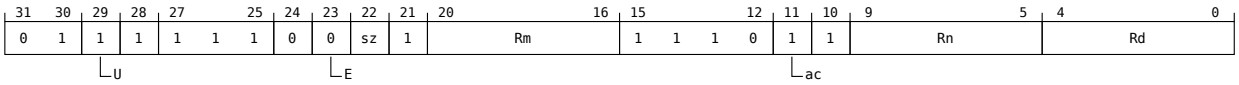
Encoding

```
FACGE <Hd>, <Hn>, <Hm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 16;
constant integer datasize = esize;
constant integer elements = 1;
```

Scalar single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

```
FACGE <V><d>, <V><n>, <V><m>
```

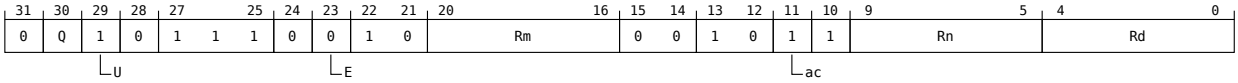
Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer esize = 32 << UInt(sz);
constant integer datasize = esize;
constant integer elements = 1;
```

Vector half-precision (FEAT_AdvSIMD && FEAT_FP16)



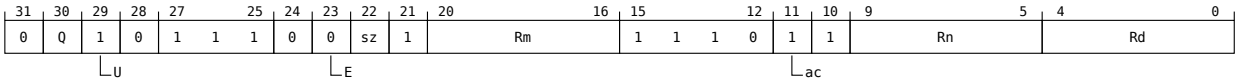
Encoding

```
FACGE <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Vector single-precision and double-precision (FEAT_AdvSIMD)



Encoding

```
FACGE <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Hm**> Is the 16-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**V**> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**n**> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<**m**> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> For the "Vector half-precision" variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the "Vector single-precision and double-precision" variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];

bits(esize) element1;
bits(esize) element2;
boolean test_passed;
bits(datasize) result;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    element1 = FPAbs(element1, FPCR);
    element2 = FPAbs(element2, FPCR);
    test_passed = FPCompareGE(element1, element2, FPCR);
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.41 FACGT

Floating-point absolute compare greater than (vector)

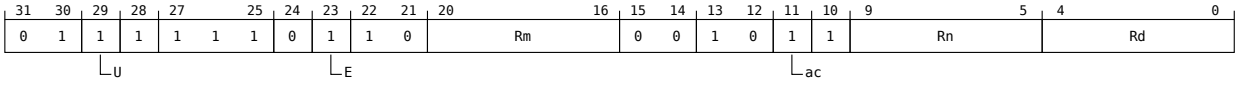
This instruction compares the absolute value of each vector element in the first source SIMD&FP register with the absolute value of the corresponding vector element in the second source SIMD&FP register and if the first value is greater than the second value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision , Scalar single-precision and double-precision , Vector half-precision and Vector single-precision and double-precision

Scalar half-precision
(FEAT_AdvSIMD && FEAT_FP16)



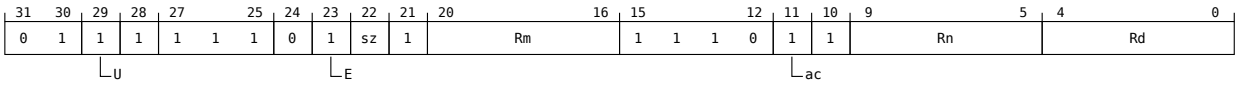
Encoding

```
FACGT <Hd>, <Hn>, <Hm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 16;
constant integer datasize = esize;
constant integer elements = 1;
```

Scalar single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

```
FACGT <V><d>, <V><n>, <V><m>
```

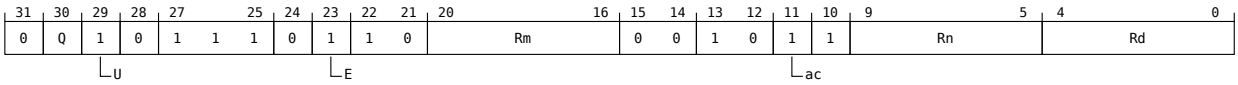
Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer esize = 32 << UInt(sz);
constant integer datasize = esize;
constant integer elements = 1;
```

Vector half-precision (FEAT_AdvSIMD && FEAT_FP16)



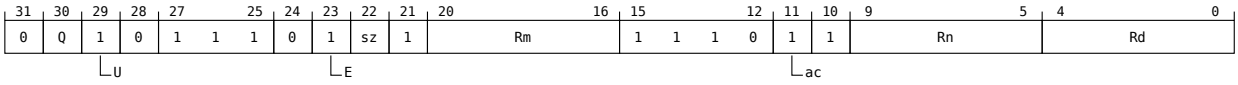
Encoding

```
FACGT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Vector single-precision and double-precision (FEAT_AdvSIMD)



Encoding

```
FACGT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Hm**> Is the 16-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**V**> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**n**> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<**m**> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> For the "Vector half-precision" variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the "Vector single-precision and double-precision" variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];

bits(esize) element1;
bits(esize) element2;
boolean test_passed;
bits(datasize) result;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    element1 = FPAbs(element1, FPCR);
    element2 = FPAbs(element2, FPCR);
    test_passed = FPCompareGT(element1, element2, FPCR);
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.42 FADD (vector)

Floating-point add (vector)

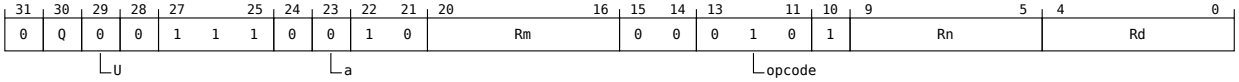
This instruction adds corresponding vector elements in the two source SIMD&FP registers, writes the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(FEAT_AdvSIMD && FEAT_FP16)



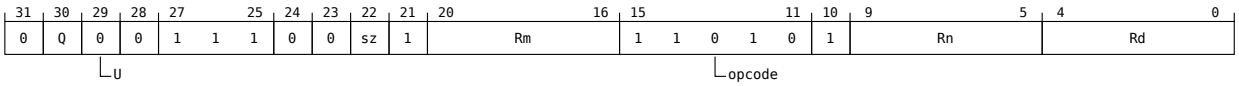
Encoding

```
FADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

```
FADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> For the "Half-precision" variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the "Single-precision and double-precision" variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPAdd(element1, element2, FPCR);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

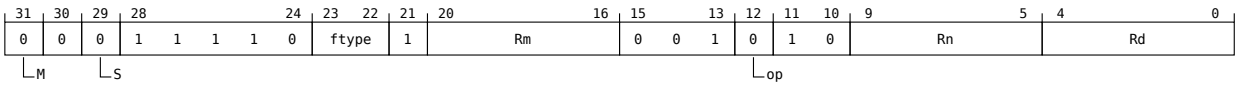
C7.2.43 FADD (scalar)

Floating-point add (scalar)

This instruction adds the floating-point values of the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Encoding for the Half-precision variant
(FEAT_FP16) Applies when `(ftype == 11)`

```
FADD <Hd>, <Hn>, <Hm>
```

Encoding for the Single-precision variant
(FEAT_FP) Applies when `(ftype == 00)`

```
FADD <Sd>, <Sn>, <Sm>
```

Encoding for the Double-precision variant
(FEAT_FP) Applies when `(ftype == 01)`

```
FADD <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if ftype == '10' then EndOfDecode(Decode_UNDEF);
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 8 << UInt(ftype EOR '10');
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Hm**> Is the 16-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<**Sd**> Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Sn**> Is the 32-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Sm**> Is the 32-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<**Dd**> Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Dn**> Is the 64-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**Dm**> Is the 64-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(esize) operand1 = V[n, esize];
constant bits(esize) operand2 = V[m, esize];

V[d, esize] = FPAdd(operand1, operand2, FPCR);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.44 FADDP (scalar)

Floating-point add pair of elements (scalar)

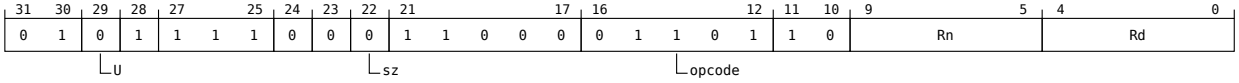
This instruction adds two floating-point vector elements in the source SIMD&FP register and writes the scalar result into the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(FEAT_AdvSIMD && FEAT_FP16)



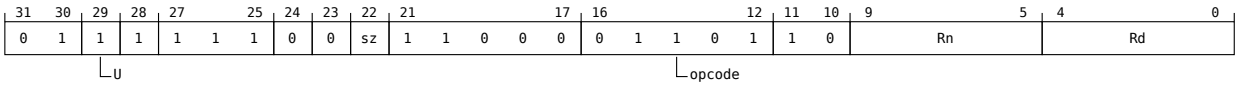
Encoding

```
FADDP H<d>, <Vn>.2H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
if sz == '1' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 16;
constant integer datasize = 32;
```

Single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

```
FADDP <V><d>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 32 << UInt(sz);
constant integer datasize = esize * 2;
```

Assembler Symbols

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<**V**> Is the destination width specifier, encoded in "sz":

sz	<V>
0	S
1	D

<**T**> Is the source arrangement specifier, encoded in "sz":

sz	<T>
0	2S
1	2D

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
constant bits(esize) element1 = Elem[operand, 0, esize];
constant bits(esize) element2 = Elem[operand, 1, esize];
V[d, esize] = FPAdd(element1, element2, FPCR);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.45 FADDP (vector)

Floating-point add pairwise (vector)

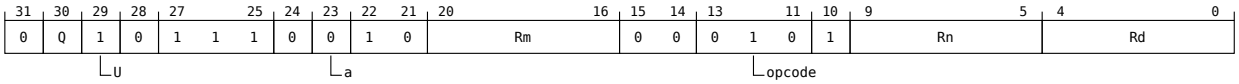
This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements from the concatenated vector, adds each pair of values together, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(FEAT_AdvSIMD && FEAT_FP16)



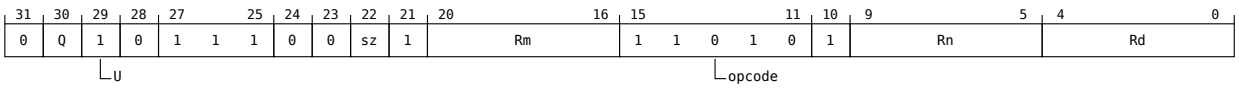
Encoding

```
FADDP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

```
FADDP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 32 << UInt(sz);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> For the "Half-precision" variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the "Single-precision and double-precision" variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
constant bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[concat, 2*e, esize];
    element2 = Elem[concat, (2*e)+1, esize];
    Elem[result, e, esize] = FPAdd(element1, element2, FPCR);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.46 FCADD

Floating-point complex add

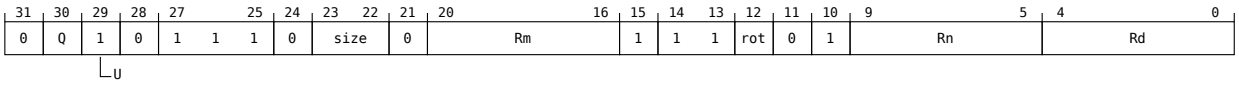
This instruction operates on complex numbers that are represented in SIMD&FP registers as pairs of elements, with the more significant element holding the imaginary part of the number and the less significant element holding the real part of the number. Each element holds a floating-point value. It performs the following calculation on the corresponding complex number element pairs from the two source registers:

- Considering the complex number from the second source register on an Argand diagram, the number is rotated counterclockwise by 90 or 270 degrees.
- The rotated complex number is added to the complex number from the first source register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector
(FEAT_FCMA)



Encoding

```
FCADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>, #<rotate>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FCMA) then EndOfDecode(Decode_UNDEF);
if size == '00' then EndOfDecode(Decode_UNDEF);
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);
if Q == '0' && size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	Q	<T>
10	1	4S
11	0	RESERVED
11	1	2D

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

<**rotate**> Is the rotation, encoded in “rot”:

rot	<rotate>
0	90
1	270

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element3;

for e = 0 to (elements DIV 2)-1
    case rot of
        when '0'
            element1 = FPNeg(Elem[operand2, e*2+1, esize], FPCR);
            element3 = Elem[operand2, e*2, esize];
        when '1'
            element1 = Elem[operand2, e*2+1, esize];
            element3 = FPNeg(Elem[operand2, e*2, esize], FPCR);
    Elem[result, e*2,   esize] = FPAdd(Elem[operand1, e*2, esize], element1, FPCR);
    Elem[result, e*2+1, esize] = FPAdd(Elem[operand1, e*2+1, esize], element3, FPCR);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.47 FCCMP

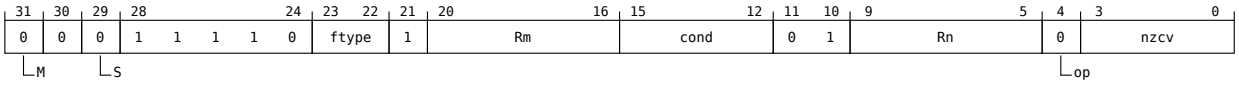
Floating-point conditional quiet compare (scalar)

This instruction compares the two SIMD&FP source register values and writes the result to the PSTATE.{N, Z, C, V} flags. If the condition does not pass, then the PSTATE.{N, Z, C, V} flags are set to the flag bit specifier.

This instruction raises an Invalid Operation floating-point exception if either or both of the operands is a signaling NaN.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Encoding for the Half-precision variant
(FEAT_FP16) Applies when `(ftype == 11)`

```
FCCMP <Hn>, <Hm>, #<nzcv>, <cond>
```

Encoding for the Single-precision variant
(FEAT_FP) Applies when `(ftype == 00)`

```
FCCMP <Sn>, <Sm>, #<nzcv>, <cond>
```

Encoding for the Double-precision variant
(FEAT_FP) Applies when `(ftype == 01)`

```
FCCMP <Dn>, <Dm>, #<nzcv>, <cond>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if ftype == '10' then EndOfDecode(Decode_UNDEF);
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);

constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 8 << UInt(ftype EOR '10');
constant bits(4) condition = cond;
bits(4) flags = nzcv;
```

Assembler Symbols

<**Hn**> Is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Hm**> Is the 16-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<**nzcv**> Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the “nzcv” field.

<**cond**> Is one of the standard conditions, encoded in the standard way, and encoded in “cond”:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

cond	<cond>
0000	EQ
0001	NE
0010	CS
0011	CC
0100	MI
0101	PL
0110	VS
0111	VC
1000	HI
1001	LS
1010	GE
1011	LT
1100	GT
1101	LE
1110	AL
1111	NV

<**Sn**> Is the 32-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Sm**> Is the 32-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<**Dn**> Is the 64-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Dm**> Is the 64-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];

if ConditionHolds(condition) then
    constant boolean signal_all_nans = FALSE;
    flags = FPCompare(operand1, operand2, signal_all_nans, FPCR);

PSTATE.<N,Z,C,V> = flags;
```

Operational information

The IEEE 754 standard specifies that the result of a comparison is precisely one of $<$, $==$, $>$ or unordered. If either or both of the operands is a NaN, they are unordered, and all three of (Operand1 $<$ Operand2), (Operand1 $==$ Operand2) and (Operand1 $>$ Operand2) are false. An unordered comparison sets the PSTATE condition flags to N=0, Z=0, C=1, and V=1.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.48 FCCMPE

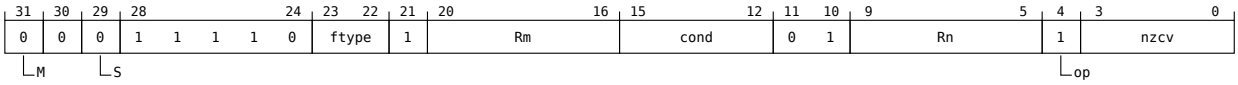
Floating-point conditional signaling compare (scalar)

This instruction compares the two SIMD&FP source register values and writes the result to the PSTATE.{N, Z, C, V} flags. If the condition does not pass, then the PSTATE.{N, Z, C, V} flags are set to the flag bit specifier.

This instruction raises an Invalid Operation floating-point exception if either or both of the operands is any type of NaN.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Encoding for the Half-precision variant
(FEAT_FP16) Applies when `(ftype == 11)`

```
FCCMPE <Hn>, <Hm>, #<nzcv>, <cond>
```

Encoding for the Single-precision variant
(FEAT_FP) Applies when `(ftype == 00)`

```
FCCMPE <Sn>, <Sm>, #<nzcv>, <cond>
```

Encoding for the Double-precision variant
(FEAT_FP) Applies when `(ftype == 01)`

```
FCCMPE <Dn>, <Dm>, #<nzcv>, <cond>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if ftype == '10' then EndOfDecode(Decode_UNDEF);
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);

constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 8 << UInt(ftype EOR '10');
constant bits(4) condition = cond;
bits(4) flags = nzcv;
```

Assembler Symbols

<**Hn**> Is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Hm**> Is the 16-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<**nzcv**> Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the “nzcv” field.

<**cond**> Is one of the standard conditions, encoded in the standard way, and encoded in “cond”:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

cond	<cond>
0000	EQ
0001	NE
0010	CS
0011	CC
0100	MI
0101	PL
0110	VS
0111	VC
1000	HI
1001	LS
1010	GE
1011	LT
1100	GT
1101	LE
1110	AL
1111	NV

<**Sn**> Is the 32-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Sm**> Is the 32-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<**Dn**> Is the 64-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Dm**> Is the 64-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];

if ConditionHolds(condition) then
    constant boolean signal_all_nans = TRUE;
    flags = FPCompare(operand1, operand2, signal_all_nans, FPCR);

PSTATE.<N,Z,C,V> = flags;
```

Operational information

The IEEE 754 standard specifies that the result of a comparison is precisely one of $<$, $==$, $>$ or unordered. If either or both of the operands is a NaN, they are unordered, and all three of (Operand1 $<$ Operand2), (Operand1 $==$ Operand2) and (Operand1 $>$ Operand2) are false. An unordered comparison sets the PSTATE condition flags to N=0, Z=0, C=1, and V=1.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.49 FCMEQ (register)

Floating-point compare equal (vector)

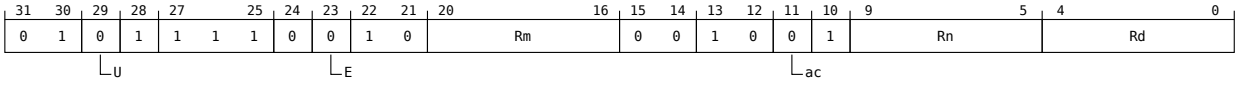
This instruction compares each floating-point value from the first source SIMD&FP register, with the corresponding floating-point value from the second source SIMD&FP register, and if the comparison is equal sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision , Scalar single-precision and double-precision , Vector half-precision and Vector single-precision and double-precision

Scalar half-precision
(FEAT_AdvSIMD && FEAT_FP16)



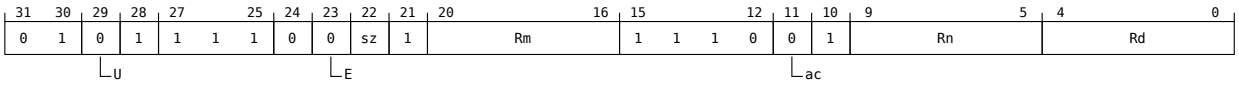
Encoding

```
FCMEQ <Hd>, <Hn>, <Hm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 16;
constant integer datasize = esize;
constant integer elements = 1;
```

Scalar single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

```
FCMEQ <V><d>, <V><n>, <V><m>
```

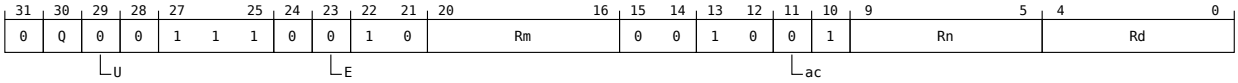
Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer esize = 32 << UInt(sz);
constant integer datasize = esize;
constant integer elements = 1;
```

Vector half-precision (FEAT_AdvSIMD && FEAT_FP16)



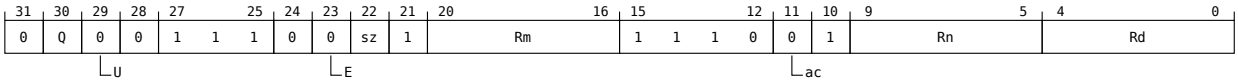
Encoding

```
FCMEQ <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Vector single-precision and double-precision (FEAT_AdvSIMD)



Encoding

```
FCMEQ <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Hm**> Is the 16-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**V**> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**n**> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<**m**> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> For the "Vector half-precision" variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the "Vector single-precision and double-precision" variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];

bits(esize) element1;
bits(esize) element2;
boolean test_passed;
bits(datasize) result;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    test_passed = FPCompareEQ(element1, element2, FPCR);
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.50 FCMEQ (zero)

Floating-point compare equal to zero (vector)

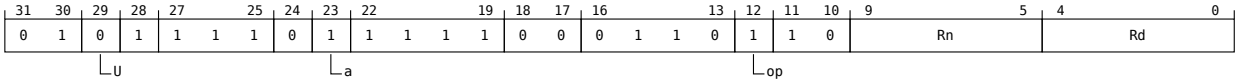
This instruction reads each floating-point value in the source SIMD&FP register and if the value is equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision , Scalar single-precision and double-precision , Vector half-precision and Vector single-precision and double-precision

Scalar half-precision
(FEAT_AdvSIMD && FEAT_FP16)



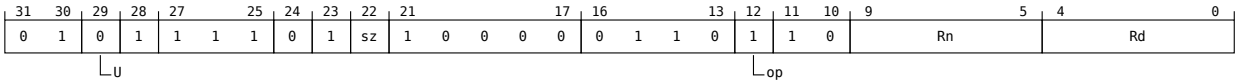
Encoding

```
FCMEQ <Hd>, <Hn>, #0.0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 16;
constant integer datasize = esize;
constant integer elements = 1;
```

Scalar single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

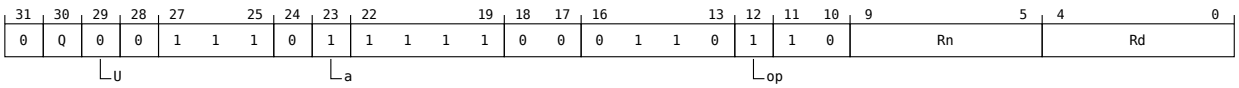
```
FCMEQ <V><d>, <V><n>, #0.0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 32 << UInt(sz);
constant integer datasize = esize;
constant integer elements = 1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Vector half-precision (FEAT_AdvSIMD && FEAT_FP16)



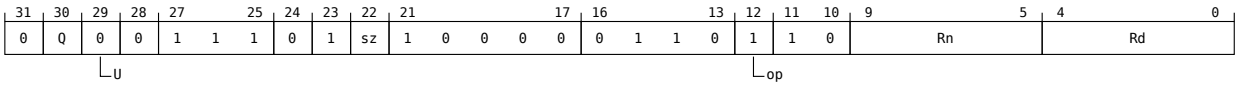
Encoding

```
FCMEQ <Vd>.<T>, <Vn>.<T>, #0.0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Vector single-precision and double-precision (FEAT_AdvSIMD)



Encoding

```
FCMEQ <Vd>.<T>, <Vn>.<T>, #0.0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<**V**> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**n**> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> For the "Vector half-precision" variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the "Vector single-precision and double-precision" variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(datasize) result;
constant bits(esize) zero = FPZero('0', esize);
bits(esize) element;
boolean test_passed;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    test_passed = FPCompareEQ(element, zero, FPCR);
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.51 FCMGE (register)

Floating-point compare greater than or equal (vector)

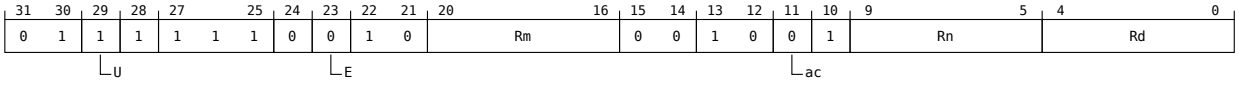
This instruction reads each floating-point value in the first source SIMD&FP register and if the value is greater than or equal to the corresponding floating-point value in the second source SIMD&FP register sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision , Scalar single-precision and double-precision , Vector half-precision and Vector single-precision and double-precision

Scalar half-precision
(FEAT_AdvSIMD && FEAT_FP16)



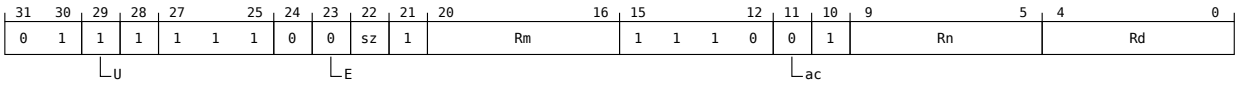
Encoding

```
FCMGE <Hd>, <Hn>, <Hm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 16;
constant integer datasize = esize;
constant integer elements = 1;
```

Scalar single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

```
FCMGE <V><d>, <V><n>, <V><m>
```

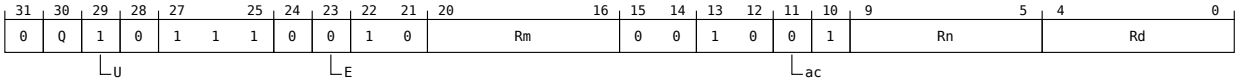
Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer esize = 32 << UInt(sz);
constant integer datasize = esize;
constant integer elements = 1;
```

Vector half-precision (FEAT_AdvSIMD && FEAT_FP16)



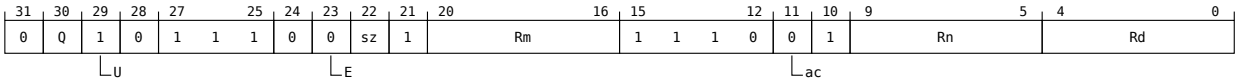
Encoding

```
FCMGE <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Vector single-precision and double-precision (FEAT_AdvSIMD)



Encoding

```
FCMGE <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Hm**> Is the 16-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**V**> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**n**> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<**m**> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> For the "Vector half-precision" variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the "Vector single-precision and double-precision" variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];

bits(esize) element1;
bits(esize) element2;
boolean test_passed;
bits(datasize) result;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    test_passed = FPCompareGE(element1, element2, FPCR);
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.52 FCMGE (zero)

Floating-point compare greater than or equal to zero (vector)

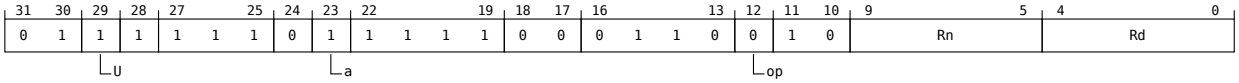
This instruction reads each floating-point value in the source SIMD&FP register and if the value is greater than or equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision , Scalar single-precision and double-precision , Vector half-precision and Vector single-precision and double-precision

Scalar half-precision
(FEAT_AdvSIMD && FEAT_FP16)



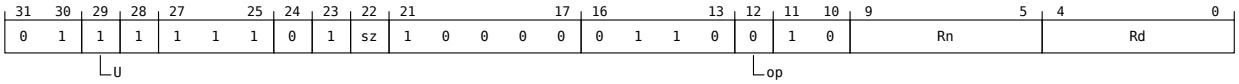
Encoding

```
FCMGE <Hd>, <Hn>, #0.0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 16;
constant integer datasize = esize;
constant integer elements = 1;
```

Scalar single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

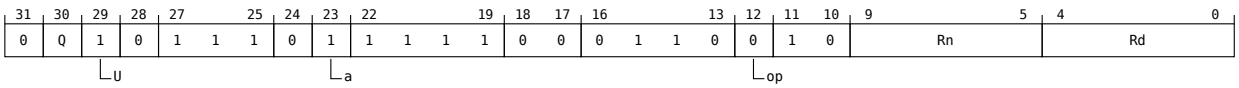
```
FCMGE <V><d>, <V><n>, #0.0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 32 << UInt(sz);
constant integer datasize = esize;
constant integer elements = 1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Vector half-precision (FEAT_AdvSIMD && FEAT_FP16)



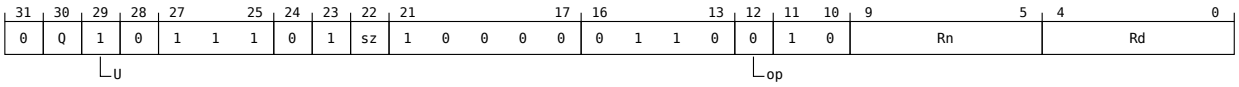
Encoding

```
FCMGE <Vd>.<T>, <Vn>.<T>, #0.0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Vector single-precision and double-precision (FEAT_AdvSIMD)



Encoding

```
FCMGE <Vd>.<T>, <Vn>.<T>, #0.0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<**V**> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**n**> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> For the "Vector half-precision" variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the "Vector single-precision and double-precision" variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(datasize) result;
constant bits(esize) zero = FPZero('0', esize);
bits(esize) element;
boolean test_passed;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    test_passed = FPCompareGE(element, zero, FPCR);
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.53 FCMGT (register)

Floating-point compare greater than (vector)

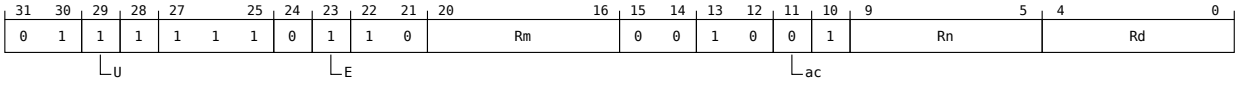
This instruction reads each floating-point value in the first source SIMD&FP register and if the value is greater than the corresponding floating-point value in the second source SIMD&FP register sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision , Scalar single-precision and double-precision , Vector half-precision and Vector single-precision and double-precision

Scalar half-precision
(FEAT_AdvSIMD && FEAT_FP16)



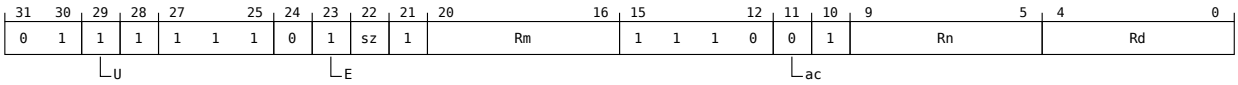
Encoding

```
FCMGT <Hd>, <Hn>, <Hm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 16;
constant integer datasize = esize;
constant integer elements = 1;
```

Scalar single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

```
FCMGT <V><d>, <V><n>, <V><m>
```

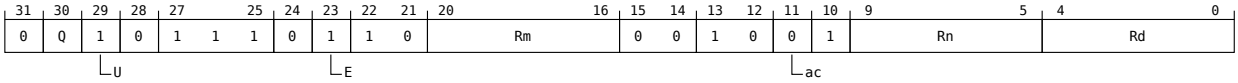
Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer esize = 32 << UInt(sz);
constant integer datasize = esize;
constant integer elements = 1;
```

Vector half-precision (FEAT_AdvSIMD && FEAT_FP16)



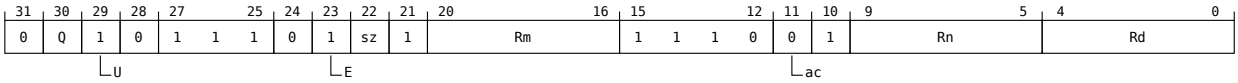
Encoding

```
FCMGT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Vector single-precision and double-precision (FEAT_AdvSIMD)



Encoding

```
FCMGT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Hm**> Is the 16-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**V**> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**n**> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<**m**> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> For the "Vector half-precision" variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the "Vector single-precision and double-precision" variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];

bits(esize) element1;
bits(esize) element2;
boolean test_passed;
bits(datasize) result;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    test_passed = FPCompareGT(element1, element2, FPCR);
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.54 FCMGT (zero)

Floating-point compare greater than zero (vector)

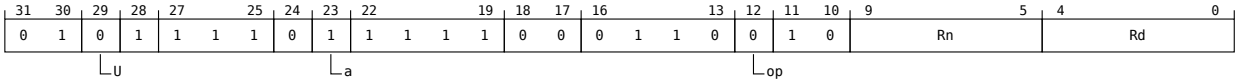
This instruction reads each floating-point value in the source SIMD&FP register and if the value is greater than zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision , Scalar single-precision and double-precision , Vector half-precision and Vector single-precision and double-precision

Scalar half-precision
(FEAT_AdvSIMD && FEAT_FP16)



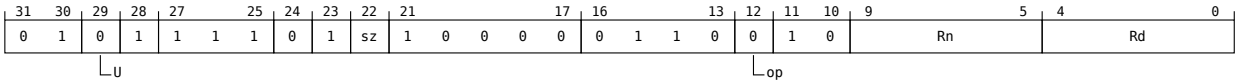
Encoding

```
FCMGT <Hd>, <Hn>, #0.0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 16;
constant integer datasize = esize;
constant integer elements = 1;
```

Scalar single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

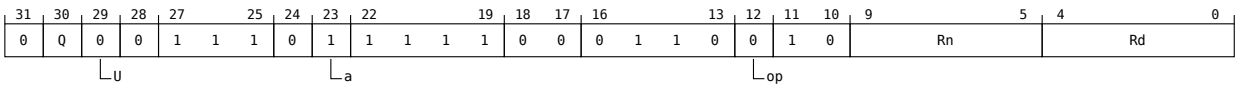
```
FCMGT <V><d>, <V><n>, #0.0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 32 << UInt(sz);
constant integer datasize = esize;
constant integer elements = 1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Vector half-precision (FEAT_AdvSIMD && FEAT_FP16)



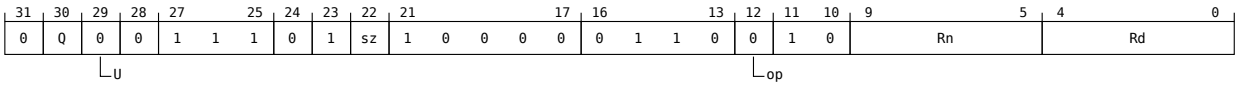
Encoding

```
FCMGT <Vd>.<T>, <Vn>.<T>, #0.0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Vector single-precision and double-precision (FEAT_AdvSIMD)



Encoding

```
FCMGT <Vd>.<T>, <Vn>.<T>, #0.0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<**V**> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**n**> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> For the "Vector half-precision" variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the "Vector single-precision and double-precision" variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(datasize) result;
constant bits(esize) zero = FPZero('0', esize);
bits(esize) element;
boolean test_passed;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    test_passed = FPCompareGT(element, zero, FPCR);
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.55 FCMLA (by element)

Floating-point complex multiply accumulate (by element)

This instruction operates on complex numbers that are represented in SIMD&FP registers as pairs of elements, with the more significant element holding the imaginary part of the number and the less significant element holding the real part of the number. Each element holds a floating-point value. It performs the following calculation on complex numbers from the first source register and the destination register with the specified complex number from the second source register:

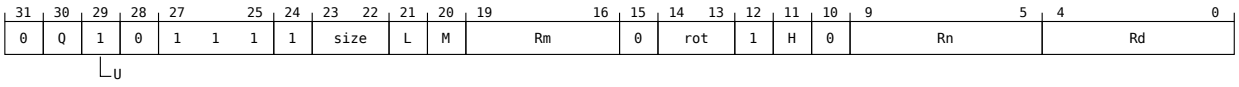
- Considering the complex number from the second source register on an Argand diagram, the number is rotated counterclockwise by 0, 90, 180, or 270 degrees.
- The two elements of the transformed complex number are multiplied by:
  - The real element of the complex number from the first source register, if the transformation was a rotation by 0 or 180 degrees.
  - The imaginary element of the complex number from the first source register, if the transformation was a rotation by 90 or 270 degrees.
- The complex number resulting from that multiplication is added to the complex number from the destination register.

The multiplication and addition operations are performed as a fused multiply-add, without any intermediate rounding.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector
(FEAT_FCMA)



Encoding

```
FCMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>], #<rotate>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FCMA) then EndOfDecode(Decode_UNDEF);
if size == '00' || size == '11' then EndOfDecode(Decode_UNDEF);
if !IsFeatureImplemented(FEAT_FP16) && size == '01' then EndOfDecode(Decode_UNDEF);
if size == '10' && (L == '1' || Q == '0') then EndOfDecode(Decode_UNDEF);
if size == '01' && H == '1' && Q == '0' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(M:Rm);
integer index;
if size == '01' then index = UInt(H:L);
if size == '10' then index = UInt(H);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	RESERVED
10	1	4S
11	x	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “M:Rm” fields.

<**Ts**> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<**index**> Is the element index, encoded in “size:H:L”:

size	<index>
00	RESERVED
01	UInt(H:L)
10	UInt(H)
11	RESERVED

<**rotate**> Is the rotation, encoded in “rot”:

rot	<rotate>
00	0
01	90
10	180
11	270

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
constant bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
for e = 0 to (elements DIV 2)-1
    bits(esize) element1;
    bits(esize) element2;
    bits(esize) element3;
    bits(esize) element4;
    case rot of
        when '00'
            element1 = Elem[operand2, index*2, esize];
            element2 = Elem[operand1, e*2, esize];
            element3 = Elem[operand2, index*2+1, esize];
            element4 = Elem[operand1, e*2, esize];
        when '01'
            element1 = FPNeg(Elem[operand2, index*2+1, esize], FPCR);
            element2 = Elem[operand1, e*2+1, esize];
            element3 = Elem[operand2, index*2, esize];
            element4 = Elem[operand1, e*2+1, esize];
        when '10'
            element1 = FPNeg(Elem[operand2, index*2, esize], FPCR);
            element2 = Elem[operand1, e*2, esize];
            element3 = FPNeg(Elem[operand2, index*2+1, esize], FPCR);
            element4 = Elem[operand1, e*2, esize];
        when '11'
            element1 = Elem[operand2, index*2+1, esize];
            element2 = Elem[operand1, e*2+1, esize];
            element3 = FPNeg(Elem[operand2, index*2, esize], FPCR);
            element4 = Elem[operand1, e*2+1, esize];

    Elem[result, e*2,   esize] = FPMulAdd(Elem[operand3, e*2, esize], element2, element1, FPCR);
    Elem[result, e*2+1, esize] = FPMulAdd(Elem[operand3, e*2+1, esize], element4, element3, FPCR);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.56 FCMLA

Floating-point complex multiply accumulate

This instruction operates on complex numbers that are represented in SIMD&FP registers as pairs of elements, with the more significant element holding the imaginary part of the number and the less significant element holding the real part of the number. Each element holds a floating-point value. It performs the following calculation on the corresponding complex number element pairs from the two source registers and the destination register:

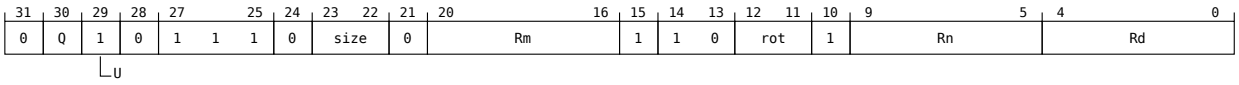
- Considering the complex number from the second source register on an Argand diagram, the number is rotated counterclockwise by 0, 90, 180, or 270 degrees.
- The two elements of the transformed complex number are multiplied by:
  - The real element of the complex number from the first source register, if the transformation was a rotation by 0 or 180 degrees.
  - The imaginary element of the complex number from the first source register, if the transformation was a rotation by 90 or 270 degrees.
- The complex number resulting from that multiplication is added to the complex number from the destination register.

The multiplication and addition operations are performed as a fused multiply-add, without any intermediate rounding.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector
(FEAT_FCMA)



Encoding

```
FCMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>, #<rotate>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FCMA) then EndOfDecode(Decode_UNDEF);
if size == '00' then EndOfDecode(Decode_UNDEF);
if !IsFeatureImplemented(FEAT_FP16) && size == '01' then EndOfDecode(Decode_UNDEF);
if Q == '0' && size == '11' then EndOfDecode(Decode_UNDEF);
constant integer esize = 8 << UInt(size);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> Is an arrangement specifier, encoded in “size:Q”:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

<**rotate**> Is the rotation, encoded in “rot”:

rot	<rotate>
00	0
01	90
10	180
11	270

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
constant bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) element3;
bits(esize) element4;

for e = 0 to (elements DIV 2)-1
    case rot of
        when '00'
            element1 = Elem[operand2, e*2, esize];
            element2 = Elem[operand1, e*2, esize];
            element3 = Elem[operand2, e*2+1, esize];
            element4 = Elem[operand1, e*2, esize];
        when '01'
            element1 = FPNeg(Elem[operand2, e*2+1, esize], FPCR);
            element2 = Elem[operand1, e*2+1, esize];
            element3 = Elem[operand2, e*2, esize];
            element4 = Elem[operand1, e*2+1, esize];
        when '10'
            element1 = FPNeg(Elem[operand2, e*2, esize], FPCR);
            element2 = Elem[operand1, e*2, esize];
            element3 = FPNeg(Elem[operand2, e*2+1, esize], FPCR);
            element4 = Elem[operand1, e*2, esize];
        when '11'
            element1 = Elem[operand2, e*2+1, esize];
            element2 = Elem[operand1, e*2+1, esize];
            element3 = FPNeg(Elem[operand2, e*2, esize], FPCR);
            element4 = Elem[operand1, e*2+1, esize];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Elem[result, e*2,   esize] = FPMulAdd(Elem[operand3, e*2, esize], element2, element1, FPCR);
        Elem[result, e*2+1, esize] = FPMulAdd(Elem[operand3, e*2+1, esize], element4, element3, FPCR);

    V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.57 FCMLE (zero)

Floating-point compare less than or equal to zero (vector)

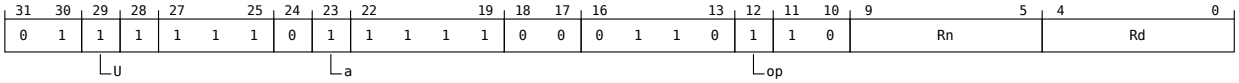
This instruction reads each floating-point value in the source SIMD&FP register and if the value is less than or equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision , Scalar single-precision and double-precision , Vector half-precision and Vector single-precision and double-precision

Scalar half-precision
(FEAT_AdvSIMD && FEAT_FP16)



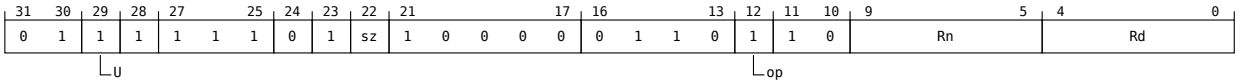
Encoding

```
FCMLE <Hd>, <Hn>, #0.0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 16;
constant integer datasize = esize;
constant integer elements = 1;
```

Scalar single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

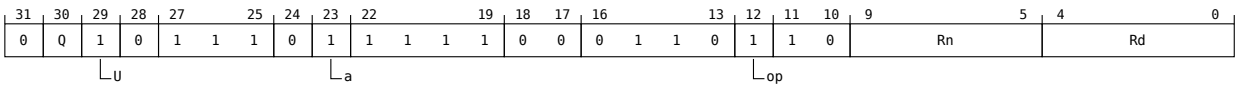
```
FCMLE <V><d>, <V><n>, #0.0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 32 << UInt(sz);
constant integer datasize = esize;
constant integer elements = 1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Vector half-precision (FEAT_AdvSIMD && FEAT_FP16)



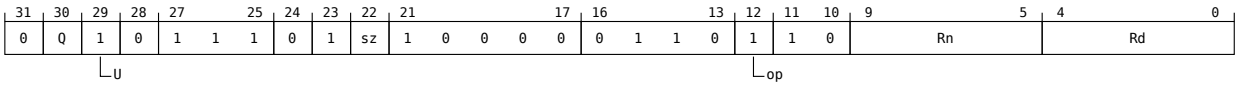
Encoding

```
FCMLE <Vd>.<T>, <Vn>.<T>, #0.0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Vector single-precision and double-precision (FEAT_AdvSIMD)



Encoding

```
FCMLE <Vd>.<T>, <Vn>.<T>, #0.0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<**V**> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**d**> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<**n**> Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> For the “Vector half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Vector single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(datasize) result;
constant bits(esize) zero = FPZero('0', esize);
bits(esize) element;
boolean test_passed;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    test_passed = FPCompareGE(zero, element, FPCR);
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.58 FCMLT (zero)

Floating-point compare less than zero (vector)

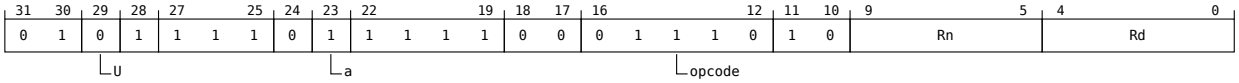
This instruction reads each floating-point value in the source SIMD&FP register and if the value is less than zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision , Scalar single-precision and double-precision , Vector half-precision and Vector single-precision and double-precision

Scalar half-precision
(FEAT_AdvSIMD && FEAT_FP16)



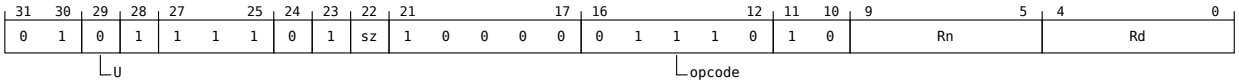
Encoding

```
FCMLT <Hd>, <Hn>, #0.0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 16;
constant integer datasize = esize;
constant integer elements = 1;
```

Scalar single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

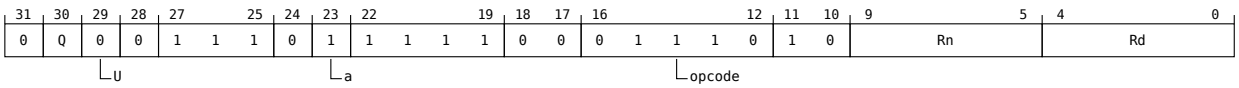
```
FCMLT <V><d>, <V><n>, #0.0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 32 << UInt(sz);
constant integer datasize = esize;
constant integer elements = 1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Vector half-precision (FEAT_AdvSIMD && FEAT_FP16)



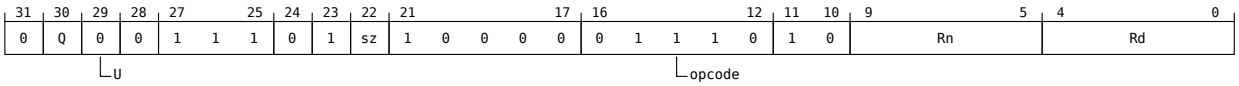
Encoding

```
FCMLT <Vd>.<T>, <Vn>.<T>, #0.0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Vector single-precision and double-precision (FEAT_AdvSIMD)



Encoding

```
FCMLT <Vd>.<T>, <Vn>.<T>, #0.0
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<**V**> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**d**> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<**n**> Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> For the “Vector half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Vector single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(datasize) result;
constant bits(esize) zero = FPZero('0', esize);
bits(esize) element;
boolean test_passed;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    test_passed = FPCompareGT(zero, element, FPCR);
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.59 FCMP

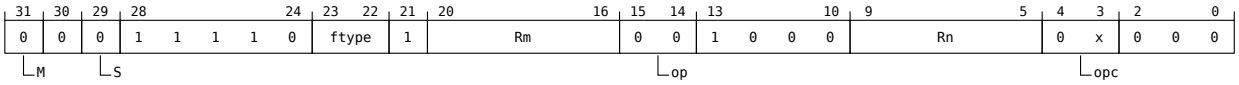
Floating-point quiet compare (scalar)

This instruction compares the two SIMD&FP source register values, or the first SIMD&FP source register value and zero. It writes the result to the PSTATE.{N, Z, C, V} flags.

This instruction raises an Invalid Operation floating-point exception if either or both of the operands is a signaling NaN.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Encoding for the Half-precision variant
(FEAT_FP16) Applies when `(ftype == 11 && opc == 00)`

```
FCMP <Hn>, <Hm>
```

Encoding for the Half-precision, zero variant
(FEAT_FP16) Applies when `(ftype == 11 && Rm == (00000) && opc == 01)`

```
FCMP <Hn>, #0.0
```

Encoding for the Single-precision variant
(FEAT_FP) Applies when `(ftype == 00 && opc == 00)`

```
FCMP <Sn>, <Sm>
```

Encoding for the Single-precision, zero variant
(FEAT_FP) Applies when `(ftype == 00 && Rm == (00000) && opc == 01)`

```
FCMP <Sn>, #0.0
```

Encoding for the Double-precision variant
(FEAT_FP) Applies when `(ftype == 01 && opc == 00)`

```
FCMP <Dn>, <Dm>
```

Encoding for the Double-precision, zero variant
(FEAT_FP) Applies when `(ftype == 01 && Rm == (00000) && opc == 01)`

```
FCMP <Dn>, #0.0
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if ftype == '10' then EndOfDecode(Decode_UNDEF);
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);

constant integer n = UInt(Rn);
constant integer m = UInt(Rm);   // ignored when opc<0> == '1'

constant integer datasize = 8 << UInt(ftype EOR '10');
constant boolean signal_all_nans = FALSE;
constant boolean cmp_with_zero = (opc<0> == '1');
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<**Hn**> For the “Half-precision” variant: is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

For the “Half-precision, zero” variant: is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<**Hm**> Is the 16-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<**Sn**> For the “Single-precision” variant: is the 32-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

For the “Single-precision, zero” variant: is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<**Sm**> Is the 32-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<**Dn**> For the “Double-precision” variant: is the 64-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

For the “Double-precision, zero” variant: is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<**Dm**> Is the 64-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = if cmp_with_zero then FPZero('0', datasize) else V[m, datasize];

PSTATE.<N,Z,C,V> = FPCompare(operand1, operand2, signal_all_nans, FPCR);
```

Operational information

The IEEE 754 standard specifies that the result of a comparison is precisely one of $<$, $==$, $>$ or unordered. If either or both of the operands is a NaN, they are unordered, and all three of (Operand1 $<$ Operand2), (Operand1 $==$ Operand2) and (Operand1 $>$ Operand2) are false. An unordered comparison sets the PSTATE condition flags to N=0, Z=0, C=1, and V=1.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.60 FCMPE

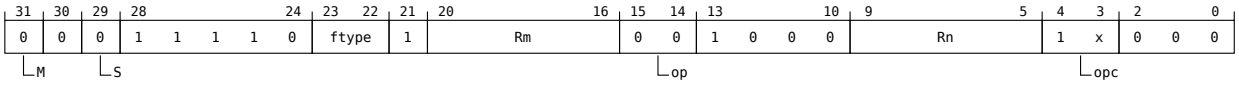
Floating-point signaling compare (scalar)

This instruction compares the two SIMD&FP source register values, or the first SIMD&FP source register value and zero. It writes the result to the PSTATE.{N, Z, C, V} flags.

This instruction raises an Invalid Operation floating-point exception if either or both of the operands is any type of NaN.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Encoding for the Half-precision variant
(FEAT_FP16) Applies when `(ftype == 11 && opc == 10)`

```
FCMPE <Hn>, <Hm>
```

Encoding for the Half-precision, zero variant
(FEAT_FP16) Applies when `(ftype == 11 && Rm == (00000) && opc == 11)`

```
FCMPE <Hn>, #0.0
```

Encoding for the Single-precision variant
(FEAT_FP) Applies when `(ftype == 00 && opc == 10)`

```
FCMPE <Sn>, <Sm>
```

Encoding for the Single-precision, zero variant
(FEAT_FP) Applies when `(ftype == 00 && Rm == (00000) && opc == 11)`

```
FCMPE <Sn>, #0.0
```

Encoding for the Double-precision variant
(FEAT_FP) Applies when `(ftype == 01 && opc == 10)`

```
FCMPE <Dn>, <Dm>
```

Encoding for the Double-precision, zero variant
(FEAT_FP) Applies when `(ftype == 01 && Rm == (00000) && opc == 11)`

```
FCMPE <Dn>, #0.0
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if ftype == '10' then EndOfDecode(Decode_UNDEF);
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);

constant integer n = UInt(Rn);
constant integer m = UInt(Rm);   // ignored when opc<0> == '1'

constant integer datasize = 8 << UInt(ftype EOR '10');
constant boolean signal_all_nans = TRUE;
constant boolean cmp_with_zero = (opc<0> == '1');
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<**Hn**> For the “Half-precision” variant: is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

For the “Half-precision, zero” variant: is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<**Hm**> Is the 16-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<**Sn**> For the “Single-precision” variant: is the 32-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

For the “Single-precision, zero” variant: is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<**Sm**> Is the 32-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<**Dn**> For the “Double-precision” variant: is the 64-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

For the “Double-precision, zero” variant: is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<**Dm**> Is the 64-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = if cmp_with_zero then FPZero('0', datasize) else V[m, datasize];

PSTATE.<N,Z,C,V> = FPCompare(operand1, operand2, signal_all_nans, FPCR);
```

Operational information

The IEEE 754 standard specifies that the result of a comparison is precisely one of $<$, $==$, $>$ or unordered. If either or both of the operands is a NaN, they are unordered, and all three of (Operand1 $<$ Operand2), (Operand1 $==$ Operand2) and (Operand1 $>$ Operand2) are false. An unordered comparison sets the PSTATE condition flags to N=0, Z=0, C=1, and V=1.

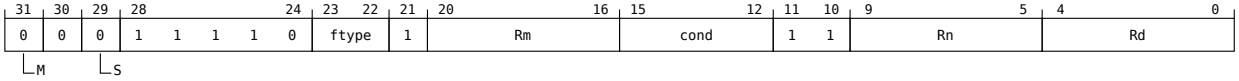
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.61 FCSEL

Floating-point conditional select (scalar)

This instruction allows the SIMD&FP destination register to take the value from either one or the other of two SIMD&FP source registers. If the condition passes, the first SIMD&FP source register value is taken, otherwise the second SIMD&FP source register value is taken.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Encoding for the Half-precision variant
(FEAT_FP16) Applies when `(ftype == 11)`

```
FCSEL <Hd>, <Hn>, <Hm>, <cond>
```

Encoding for the Single-precision variant
(FEAT_FP) Applies when `(ftype == 00)`

```
FCSEL <Sd>, <Sn>, <Sm>, <cond>
```

Encoding for the Double-precision variant
(FEAT_FP) Applies when `(ftype == 01)`

```
FCSEL <Dd>, <Dn>, <Dm>, <cond>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if ftype == '10' then EndOfDecode(Decode_UNDEF);
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer datasize = 8 << UInt(ftype EOR '10');
constant bits(4) condition = cond;
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Hm**> Is the 16-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<**cond**> Is one of the standard conditions, encoded in the standard way, and encoded in “cond”:

cond	<cond>
0000	EQ
0001	NE
0010	CS

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

cond	<cond>
0011	CC
0100	MI
0101	PL
0110	VS
0111	VC
1000	HI
1001	LS
1010	GE
1011	LT
1100	GT
1101	LE
1110	AL
1111	NV

<**Sd**> Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Sn**> Is the 32-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Sm**> Is the 32-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<**Dd**> Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Dn**> Is the 64-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Dm**> Is the 64-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
bits(datasize) result;

constant boolean condition_holds = ConditionHolds(condition);
result = if condition_holds then V[n, datasize] else V[m, datasize];

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

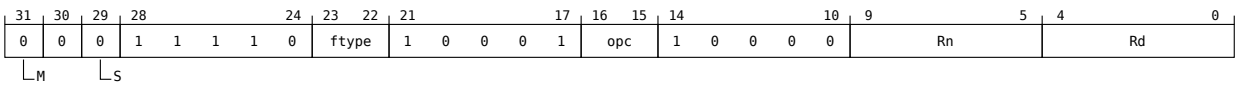
C7.2.62 FCVT

Floating-point convert precision (scalar)

This instruction converts the floating-point value in the SIMD&FP source register to the precision for the destination register data type using the rounding mode that is determined by the FPCR and writes the result to the SIMD&FP destination register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Floating-point
(FEAT_FP)



Encoding for the Half-precision to single-precision variant Applies when `(ftype == 11 && opc == 00)`

```
FCVT <Sd>, <Hn>
```

Encoding for the Half-precision to double-precision variant Applies when `(ftype == 11 && opc == 01)`

```
FCVT <Dd>, <Hn>
```

Encoding for the Single-precision to half-precision variant Applies when `(ftype == 00 && opc == 11)`

```
FCVT <Hd>, <Sn>
```

Encoding for the Single-precision to double-precision variant Applies when `(ftype == 00 && opc == 01)`

```
FCVT <Dd>, <Sn>
```

Encoding for the Double-precision to half-precision variant Applies when `(ftype == 01 && opc == 11)`

```
FCVT <Hd>, <Dn>
```

Encoding for the Double-precision to single-precision variant Applies when `(ftype == 01 && opc == 00)`

```
FCVT <Sd>, <Dn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if ftype == opc || ftype == '10' || opc == '10' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer srcsize = 8 << UInt(ftype EOR '10');
constant integer dstsize = 8 << UInt(opc EOR '10');
```

Assembler Symbols

<**Sd**> Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<**Dd**> Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Sn**> Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<**Dn**> Is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(srcsize) operand = V[n, srcsize];

V[d, dstsize] = FPConvert(operand, FPCR, dstsize);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.63 FCVTAS (vector)

Floating-point convert to signed integer, rounding to nearest with ties to away (vector)

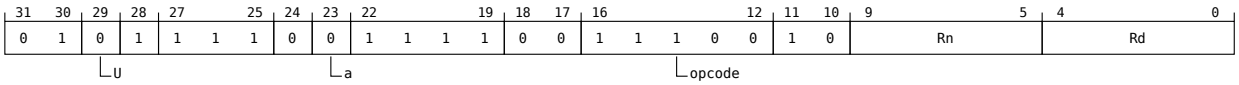
This instruction converts each element in a vector from a floating-point value to a signed integer value using the Round to Nearest with Ties to Away rounding mode and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision , Scalar single-precision and double-precision , Vector half-precision and Vector single-precision and double-precision

Scalar half-precision
(FEAT_AdvSIMD && FEAT_FP16)



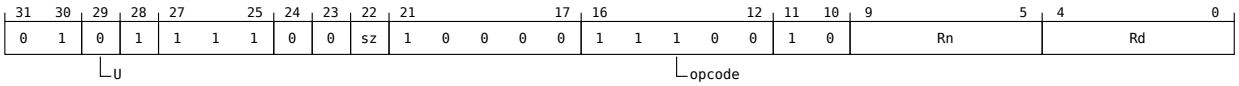
Encoding

```
FCVTAS <Hd>, <Hn>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 16;
constant integer datasize = esize;
constant integer elements = 1;
constant FPRounding rounding = FPRounding_TIEAWAY;
constant boolean unsigned = FALSE;
```

Scalar single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

```
FCVTAS <V><d>, <V><n>
```

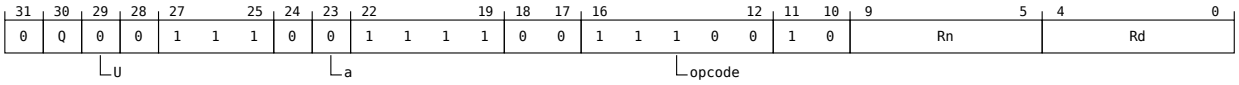
Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 32 << UInt(sz);
constant integer datasize = esize;
constant integer elements = 1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant FPRounding rounding = FPRounding_TIEAWAY;
constant boolean unsigned = FALSE;
```

Vector half-precision (FEAT_AdvSIMD && FEAT_FP16)



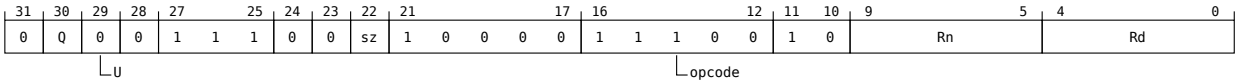
Encoding

```
FCVTAS <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
constant FPRounding rounding = FPRounding_TIEAWAY;
constant boolean unsigned = FALSE;
```

Vector single-precision and double-precision (FEAT_AdvSIMD)



Encoding

```
FCVTAS <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
constant FPRounding rounding = FPRounding_TIEAWAY;
constant boolean unsigned = FALSE;
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<**V**> Is a width specifier, encoded in “sz”:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	<V>
0	S
1	D

<**d**> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<**n**> Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> For the “Vector half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Vector single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(datasize) operand = V[n, datasize];

bits(datasize) result;
constant integer fracbits = 0;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, fracbits, unsigned, FPCR, rounding, esize);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

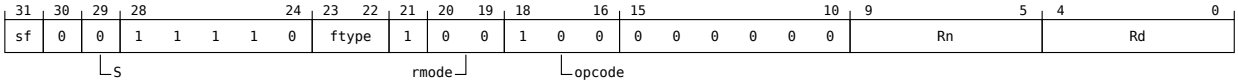
C7.2.64 FCVTAS (scalar)

Floating-point convert to signed integer, rounding to nearest with ties to away (scalar)

This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round to Nearest with Ties to Away rounding mode, and writes the result to the general-purpose destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Encoding for the Half-precision to 32-bit variant
(FEAT_FP16) Applies when `(sf == 0 && ftype == 11)`

```
FCVTAS <Wd>, <Hn>
```

Encoding for the Half-precision to 64-bit variant
(FEAT_FP16) Applies when `(sf == 1 && ftype == 11)`

```
FCVTAS <Xd>, <Hn>
```

Encoding for the Single-precision to 32-bit variant
(FEAT_FP) Applies when `(sf == 0 && ftype == 00)`

```
FCVTAS <Wd>, <Sn>
```

Encoding for the Single-precision to 64-bit variant
(FEAT_FP) Applies when `(sf == 1 && ftype == 00)`

```
FCVTAS <Xd>, <Sn>
```

Encoding for the Double-precision to 32-bit variant
(FEAT_FP) Applies when `(sf == 0 && ftype == 01)`

```
FCVTAS <Wd>, <Dn>
```

Encoding for the Double-precision to 64-bit variant
(FEAT_FP) Applies when `(sf == 1 && ftype == 01)`

```
FCVTAS <Xd>, <Dn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if ftype == '10' then EndOfDecode(Decode_UNDEF);
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer intsize = 32 << UInt(sf);
constant integer fltsize = 8 << UInt(ftype EOR '10');

constant FPRounding rounding = FPRounding_TIEAWAY;
constant boolean unsigned    = FALSE;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Hn**> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Sn**> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<**Dn**> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(fltsize) fltval = V[n, fltsize];
constant integer fracbits = 0;

X[d, intsize] = FPToFixed(fltval, fracbits, unsigned, FPCR, rounding, intsize);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.65 FCVTAU (vector)

Floating-point convert to unsigned integer, rounding to nearest with ties to away (vector)

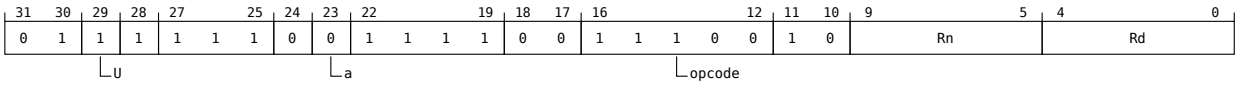
This instruction converts each element in a vector from a floating-point value to an unsigned integer value using the Round to Nearest with Ties to Away rounding mode and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision , Scalar single-precision and double-precision , Vector half-precision and Vector single-precision and double-precision

Scalar half-precision
(FEAT_AdvSIMD && FEAT_FP16)



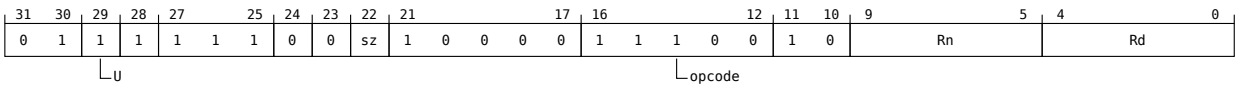
Encoding

```
FCVTAU <Hd>, <Hn>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 16;
constant integer datasize = esize;
constant integer elements = 1;
constant FPRounding rounding = FPRounding_TIEAWAY;
constant boolean unsigned = TRUE;
```

Scalar single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

```
FCVTAU <V><d>, <V><n>
```

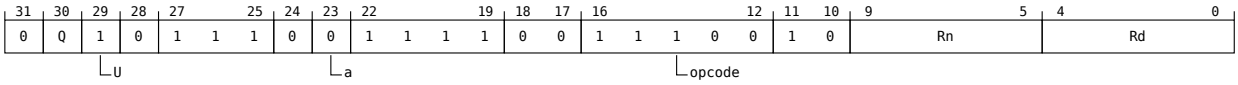
Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 32 << UInt(sz);
constant integer datasize = esize;
constant integer elements = 1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant FPRounding rounding = FPRounding_TIEAWAY;
constant boolean unsigned = TRUE;
```

Vector half-precision (FEAT_AdvSIMD && FEAT_FP16)



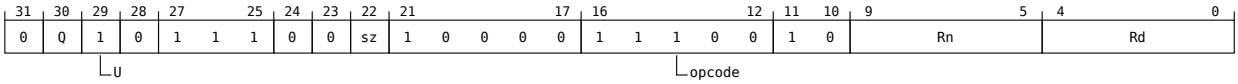
Encoding

```
FCVTAU <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
constant FPRounding rounding = FPRounding_TIEAWAY;
constant boolean unsigned = TRUE;
```

Vector single-precision and double-precision (FEAT_AdvSIMD)



Encoding

```
FCVTAU <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
constant FPRounding rounding = FPRounding_TIEAWAY;
constant boolean unsigned = TRUE;
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<**V**> Is a width specifier, encoded in “sz”:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	<V>
0	S
1	D

<**d**> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<**n**> Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> For the “Vector half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Vector single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(datasize) operand = V[n, datasize];

bits(datasize) result;
constant integer fracbits = 0;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, fracbits, unsigned, FPCR, rounding, esize);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

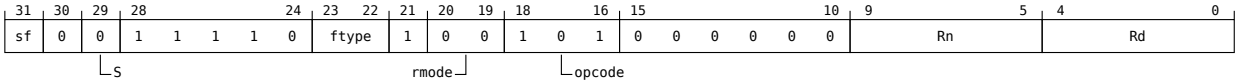
C7.2.66 FCVTAU (scalar)

Floating-point convert to unsigned integer, rounding to nearest with ties to away (scalar)

This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round to Nearest with Ties to Away rounding mode, and writes the result to the general-purpose destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Encoding for the Half-precision to 32-bit variant
(FEAT_FP16) Applies when `(sf == 0 && ftype == 11)`

```
FCVTAU <Wd>, <Hn>
```

Encoding for the Half-precision to 64-bit variant
(FEAT_FP16) Applies when `(sf == 1 && ftype == 11)`

```
FCVTAU <Xd>, <Hn>
```

Encoding for the Single-precision to 32-bit variant
(FEAT_FP) Applies when `(sf == 0 && ftype == 00)`

```
FCVTAU <Wd>, <Sn>
```

Encoding for the Single-precision to 64-bit variant
(FEAT_FP) Applies when `(sf == 1 && ftype == 00)`

```
FCVTAU <Xd>, <Sn>
```

Encoding for the Double-precision to 32-bit variant
(FEAT_FP) Applies when `(sf == 0 && ftype == 01)`

```
FCVTAU <Wd>, <Dn>
```

Encoding for the Double-precision to 64-bit variant
(FEAT_FP) Applies when `(sf == 1 && ftype == 01)`

```
FCVTAU <Xd>, <Dn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if ftype == '10' then EndOfDecode(Decode_UNDEF);
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer intsize = 32 << UInt(sf);
constant integer fltsize = 8 << UInt(ftype EOR '10');

constant FPRounding rounding = FPRounding_TIEAWAY;
constant boolean unsigned    = TRUE;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Hn**> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Sn**> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<**Dn**> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(fltsize) fltval = V[n, fltsize];
constant integer fracbits = 0;

X[d, intsize] = FPToFixed(fltval, fracbits, unsigned, FPCR, rounding, intsize);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.67 FCVTL, FCVTL2

Floating-point convert to higher precision long (vector)

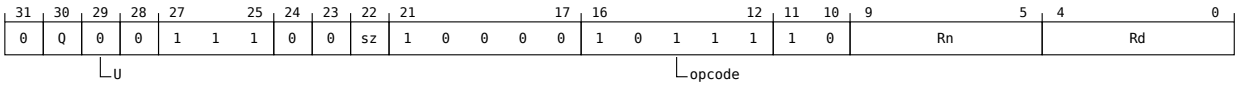
This instruction reads each element in a vector in the SIMD&FP source register, converts each value to double the precision of the source element using the rounding mode that is determined by the FPCR, and writes each result to the equivalent element of the vector in the SIMD&FP destination register.

Where the operation lengthens a 64-bit vector to a 128-bit vector, the `FCVTL2` variant operates on the elements in the top 64 bits of the source register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector single-precision and double-precision (FEAT_AdvSIMD)



Encoding

```
FCVTL{2} <Vd>.<Ta>, <Vn>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 16 << UInt(sz);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "sz":

sz	<Ta>
0	4S

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	<Ta>
1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<**Tb**> Is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<Tb>
0	0	4H
0	1	8H
1	0	2S
1	1	4S

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = Vpart[n, part, datasize];
bits(2*datasize) result;

for e = 0 to elements-1
    Elem[result, e, 2*esize] = FPConvert(Elem[operand, e, esize], FPCR, 2 * esize);

V[d, 2*datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.68 FCVTMS (vector)

Floating-point convert to signed integer, rounding toward minus infinity (vector)

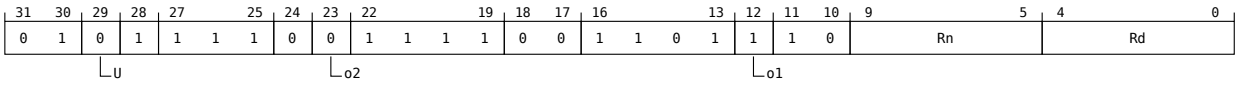
This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision , Scalar single-precision and double-precision , Vector half-precision and Vector single-precision and double-precision

Scalar half-precision
(FEAT_AdvSIMD && FEAT_FP16)



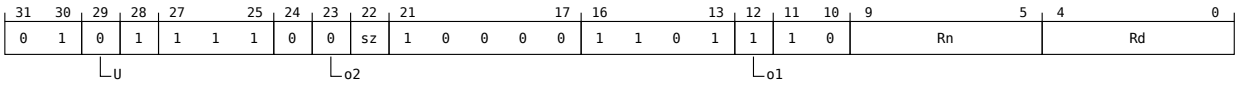
Encoding

```
FCVTMS <Hd>, <Hn>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 16;
constant integer datasize = esize;
constant integer elements = 1;
constant FPRounding rounding = FPDecodeRounding(o1:o2);
constant boolean unsigned = FALSE;
```

Scalar single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

```
FCVTMS <V><d>, <V><n>
```

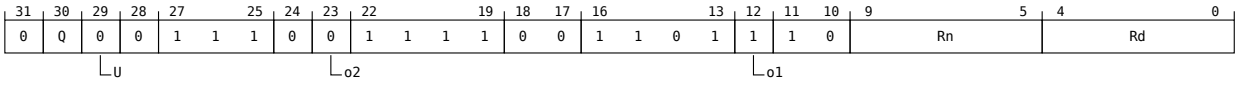
Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 32 << UInt(sz);
constant integer datasize = esize;
constant integer elements = 1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant FPRounding rounding = FPDecodeRounding(o1:o2);
constant boolean unsigned = FALSE;
```

Vector half-precision (FEAT_AdvSIMD && FEAT_FP16)



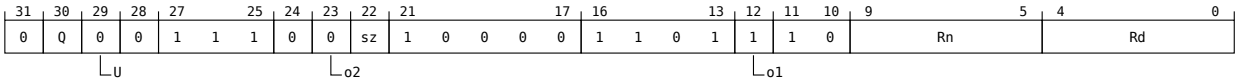
Encoding

```
FCVTMS <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
constant FPRounding rounding = FPDecodeRounding(o1:o2);
constant boolean unsigned = FALSE;
```

Vector single-precision and double-precision (FEAT_AdvSIMD)



Encoding

```
FCVTMS <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
constant FPRounding rounding = FPDecodeRounding(o1:o2);
constant boolean unsigned = FALSE;
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<**V**> Is a width specifier, encoded in “sz”:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	<V>
0	S
1	D

<**d**> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<**n**> Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> For the “Vector half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Vector single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(datasize) operand = V[n, datasize];

bits(datasize) result;
constant integer fracbits = 0;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, fracbits, unsigned, FPCR, rounding, esize);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

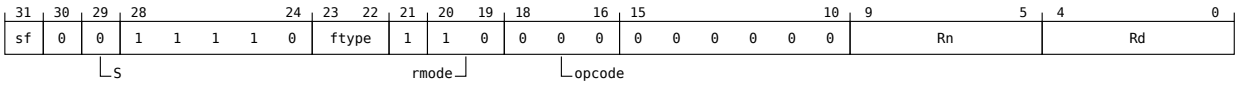
C7.2.69 FCVTMS (scalar)

Floating-point convert to signed integer, rounding toward minus infinity (scalar)

This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round towards Minus Infinity rounding mode, and writes the result to the general-purpose destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Encoding for the Half-precision to 32-bit variant
(FEAT_FP16) Applies when `(sf == 0 && ftype == 11)`

```
FCVTMS <Wd>, <Hn>
```

Encoding for the Half-precision to 64-bit variant
(FEAT_FP16) Applies when `(sf == 1 && ftype == 11)`

```
FCVTMS <Xd>, <Hn>
```

Encoding for the Single-precision to 32-bit variant
(FEAT_FP) Applies when `(sf == 0 && ftype == 00)`

```
FCVTMS <Wd>, <Sn>
```

Encoding for the Single-precision to 64-bit variant
(FEAT_FP) Applies when `(sf == 1 && ftype == 00)`

```
FCVTMS <Xd>, <Sn>
```

Encoding for the Double-precision to 32-bit variant
(FEAT_FP) Applies when `(sf == 0 && ftype == 01)`

```
FCVTMS <Wd>, <Dn>
```

Encoding for the Double-precision to 64-bit variant
(FEAT_FP) Applies when `(sf == 1 && ftype == 01)`

```
FCVTMS <Xd>, <Dn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if ftype == '10' then EndOfDecode(Decode_UNDEF);
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer intsize = 32 << UInt(sf);
constant integer fltsize = 8 << UInt(ftype EOR '10');

constant FPRounding rounding = FPRounding_NEGINF;
constant boolean unsigned    = FALSE;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Hn**> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Sn**> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<**Dn**> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(fltsize) fltval = V[n, fltsize];
constant integer fracbits = 0;

X[d, intsize] = FPToFixed(fltval, fracbits, unsigned, FPCR, rounding, intsize);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.70 FCVTMU (vector)

Floating-point convert to unsigned integer, rounding toward minus infinity (vector)

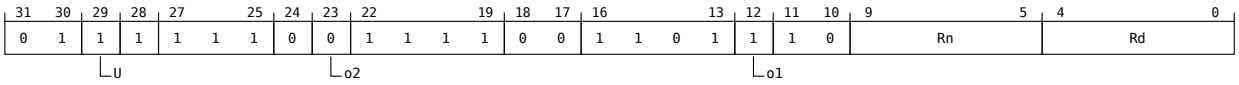
This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision , Scalar single-precision and double-precision , Vector half-precision and Vector single-precision and double-precision

Scalar half-precision
(FEAT_AdvSIMD && FEAT_FP16)



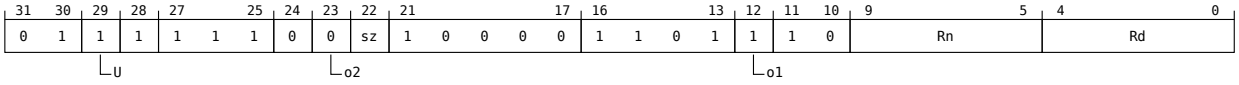
Encoding

```
FCVTMU <Hd>, <Hn>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 16;
constant integer datasize = esize;
constant integer elements = 1;
constant FPRounding rounding = FPDecodeRounding(o1:o2);
constant boolean unsigned = TRUE;
```

Scalar single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

```
FCVTMU <V><d>, <V><n>
```

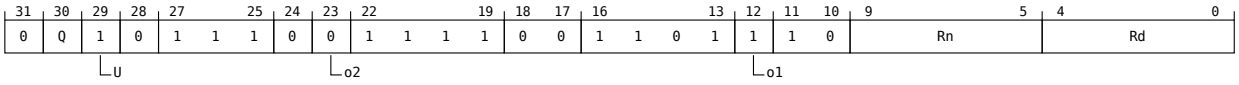
Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 32 << UInt(sz);
constant integer datasize = esize;
constant integer elements = 1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant FPRounding rounding = FPDecodeRounding(o1:o2);
constant boolean unsigned = TRUE;
```

Vector half-precision (FEAT_AdvSIMD && FEAT_FP16)



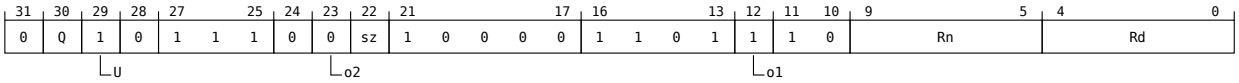
Encoding

```
FCVTMU <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
constant FPRounding rounding = FPDecodeRounding(o1:o2);
constant boolean unsigned = TRUE;
```

Vector single-precision and double-precision (FEAT_AdvSIMD)



Encoding

```
FCVTMU <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
constant FPRounding rounding = FPDecodeRounding(o1:o2);
constant boolean unsigned = TRUE;
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<**V**> Is a width specifier, encoded in “sz”:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	<V>
0	S
1	D

<**d**> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<**n**> Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> For the “Vector half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Vector single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(datasize) operand = V[n, datasize];

bits(datasize) result;
constant integer fracbits = 0;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, fracbits, unsigned, FPCR, rounding, esize);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

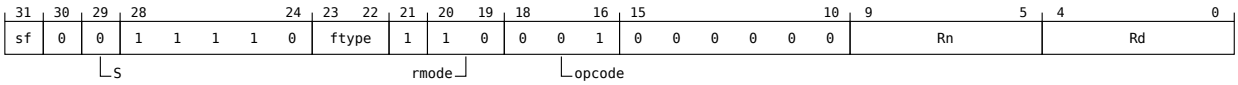
C7.2.71 FCVTMU (scalar)

Floating-point convert to unsigned integer, rounding toward minus infinity (scalar)

This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round towards Minus Infinity rounding mode, and writes the result to the general-purpose destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Encoding for the Half-precision to 32-bit variant
(FEAT_FP16) Applies when `(sf == 0 && ftype == 11)`

```
FCVTMU <Wd>, <Hn>
```

Encoding for the Half-precision to 64-bit variant
(FEAT_FP16) Applies when `(sf == 1 && ftype == 11)`

```
FCVTMU <Xd>, <Hn>
```

Encoding for the Single-precision to 32-bit variant
(FEAT_FP) Applies when `(sf == 0 && ftype == 00)`

```
FCVTMU <Wd>, <Sn>
```

Encoding for the Single-precision to 64-bit variant
(FEAT_FP) Applies when `(sf == 1 && ftype == 00)`

```
FCVTMU <Xd>, <Sn>
```

Encoding for the Double-precision to 32-bit variant
(FEAT_FP) Applies when `(sf == 0 && ftype == 01)`

```
FCVTMU <Wd>, <Dn>
```

Encoding for the Double-precision to 64-bit variant
(FEAT_FP) Applies when `(sf == 1 && ftype == 01)`

```
FCVTMU <Xd>, <Dn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if ftype == '10' then EndOfDecode(Decode_UNDEF);
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer intsize = 32 << UInt(sf);
constant integer fltsize = 8 << UInt(ftype EOR '10');

constant FPRounding rounding = FPRounding_NEGINF;
constant boolean unsigned    = TRUE;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Hn**> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Sn**> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<**Dn**> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(fltsize) fltval = V[n, fltsize];
constant integer fracbits = 0;

X[d, intsize] = FPToFixed(fltval, fracbits, unsigned, FPCR, rounding, intsize);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.72 FCVTN, FCVTN2 (double to single-precision, single to half-precision)

Floating-point convert to lower precision narrow (vector)

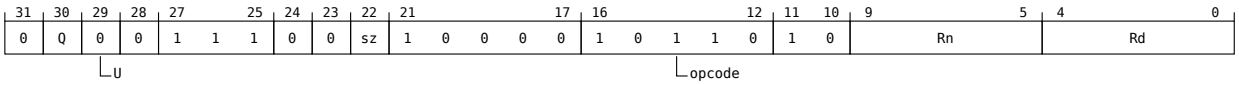
This instruction reads each vector element in the SIMD&FP source register, converts each result to half the precision of the source element, writes the final result to a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements. The rounding mode is determined by the FPCR.

FCVTN writes the vector to the lower half of the destination register and clears the upper half. FCVTN2 writes the vector to the upper half of the destination register without affecting the other bits of the register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

```
FCVTN{2} <Vd>.<Tb>, <Vn>.<Ta>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 16 << UInt(sz);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in "sz:Q":

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	Q	<Tb>
0	0	4H
0	1	8H
1	0	2S
1	1	4S

<**Vn**> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<**Ta**> Is an arrangement specifier, encoded in “sz”:

sz	<Ta>
0	4S
1	2D

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(2*datasize) operand = V[n, 2*datasize];
bits(datasize) result;

for e = 0 to elements-1
    Elem[result, e, esize] = FPConvert(Elem[operand, e, 2*esize], FPCR, esize);

Vpart[d, part, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.73 FCVTNS (vector)

Floating-point convert to signed integer, rounding to nearest with ties to even (vector)

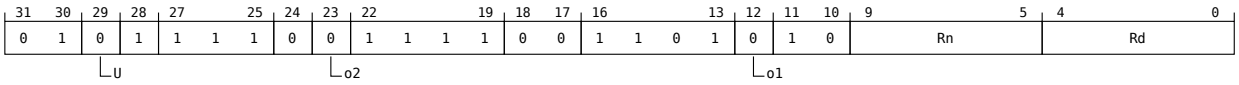
This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round to Nearest rounding mode, and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision , Scalar single-precision and double-precision , Vector half-precision and Vector single-precision and double-precision

Scalar half-precision
(FEAT_AdvSIMD && FEAT_FP16)



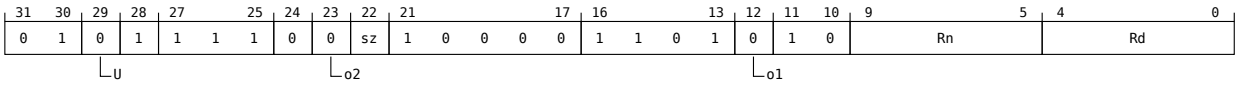
Encoding

```
FCVTNS <Hd>, <Hn>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 16;
constant integer datasize = esize;
constant integer elements = 1;
constant FPRounding rounding = FPDecodeRounding(o1:o2);
constant boolean unsigned = FALSE;
```

Scalar single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

```
FCVTNS <V><d>, <V><n>
```

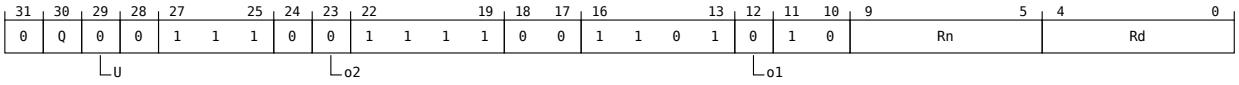
Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 32 << UInt(sz);
constant integer datasize = esize;
constant integer elements = 1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant FPRounding rounding = FPDecodeRounding(o1:o2);
constant boolean unsigned = FALSE;
```

Vector half-precision (FEAT_AdvSIMD && FEAT_FP16)



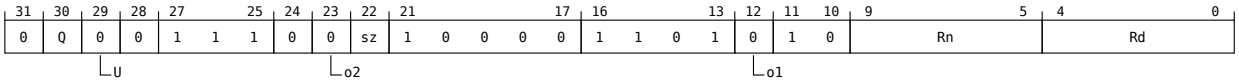
Encoding

```
FCVTNS <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
constant FPRounding rounding = FPDecodeRounding(o1:o2);
constant boolean unsigned = FALSE;
```

Vector single-precision and double-precision (FEAT_AdvSIMD)



Encoding

```
FCVTNS <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
constant FPRounding rounding = FPDecodeRounding(o1:o2);
constant boolean unsigned = FALSE;
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<**V**> Is a width specifier, encoded in “sz”:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	<V>
0	S
1	D

<**d**> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<**n**> Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> For the “Vector half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Vector single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(datasize) operand = V[n, datasize];

bits(datasize) result;
constant integer fracbits = 0;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, fracbits, unsigned, FPCR, rounding, esize);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

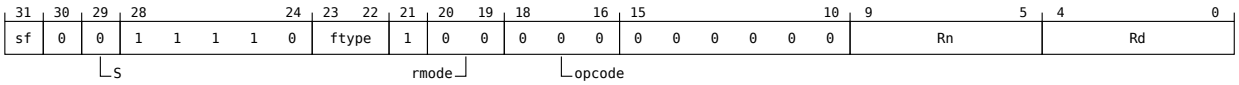
C7.2.74 FCVTNS (scalar)

Floating-point convert to signed integer, rounding to nearest with ties to even (scalar)

This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round to Nearest rounding mode, and writes the result to the general-purpose destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Encoding for the Half-precision to 32-bit variant
(FEAT_FP16) Applies when `(sf == 0 && ftype == 11)`

```
FCVTNS <Wd>, <Hn>
```

Encoding for the Half-precision to 64-bit variant
(FEAT_FP16) Applies when `(sf == 1 && ftype == 11)`

```
FCVTNS <Xd>, <Hn>
```

Encoding for the Single-precision to 32-bit variant
(FEAT_FP) Applies when `(sf == 0 && ftype == 00)`

```
FCVTNS <Wd>, <Sn>
```

Encoding for the Single-precision to 64-bit variant
(FEAT_FP) Applies when `(sf == 1 && ftype == 00)`

```
FCVTNS <Xd>, <Sn>
```

Encoding for the Double-precision to 32-bit variant
(FEAT_FP) Applies when `(sf == 0 && ftype == 01)`

```
FCVTNS <Wd>, <Dn>
```

Encoding for the Double-precision to 64-bit variant
(FEAT_FP) Applies when `(sf == 1 && ftype == 01)`

```
FCVTNS <Xd>, <Dn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if ftype == '10' then EndOfDecode(Decode_UNDEF);
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer intsize = 32 << UInt(sf);
constant integer fltsize = 8 << UInt(ftype EOR '10');

constant FPRounding rounding = FPRounding_TIEEVEN;
constant boolean unsigned    = FALSE;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Hn**> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Sn**> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<**Dn**> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(fltsize) fltval = V[n, fltsize];
constant integer fracbits = 0;

X[d, intsize] = FPToFixed(fltval, fracbits, unsigned, FPCR, rounding, intsize);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.75 FCVTNU (vector)

Floating-point convert to unsigned integer, rounding to nearest with ties to even (vector)

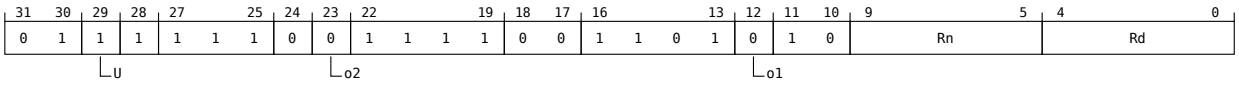
This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round to Nearest rounding mode, and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision , Scalar single-precision and double-precision , Vector half-precision and Vector single-precision and double-precision

Scalar half-precision
(FEAT_AdvSIMD && FEAT_FP16)



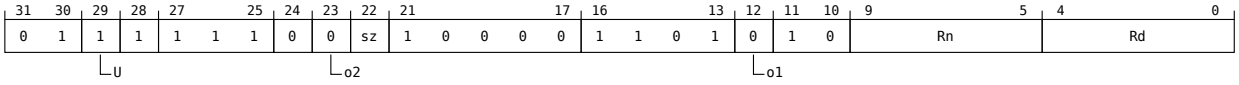
Encoding

```
FCVTNU <Hd>, <Hn>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 16;
constant integer datasize = esize;
constant integer elements = 1;
constant FPRounding rounding = FPDecodeRounding(o1:o2);
constant boolean unsigned = TRUE;
```

Scalar single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

```
FCVTNU <V><d>, <V><n>
```

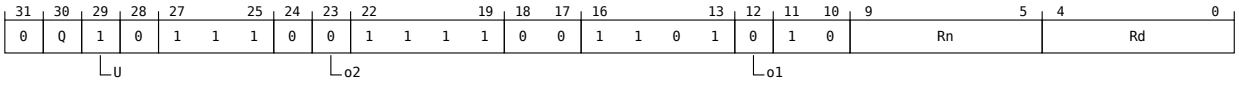
Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 32 << UInt(sz);
constant integer datasize = esize;
constant integer elements = 1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant FPRounding rounding = FPDecodeRounding(o1:o2);
constant boolean unsigned = TRUE;
```

Vector half-precision (FEAT_AdvSIMD && FEAT_FP16)



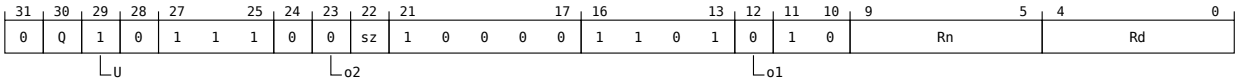
Encoding

```
FCVTNU <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
constant FPRounding rounding = FPDecodeRounding(o1:o2);
constant boolean unsigned = TRUE;
```

Vector single-precision and double-precision (FEAT_AdvSIMD)



Encoding

```
FCVTNU <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
constant FPRounding rounding = FPDecodeRounding(o1:o2);
constant boolean unsigned = TRUE;
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<**V**> Is a width specifier, encoded in “sz”:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	<V>
0	S
1	D

<**d**> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<**n**> Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> For the “Vector half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Vector single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(datasize) operand = V[n, datasize];

bits(datasize) result;
constant integer fracbits = 0;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, fracbits, unsigned, FPCR, rounding, esize);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

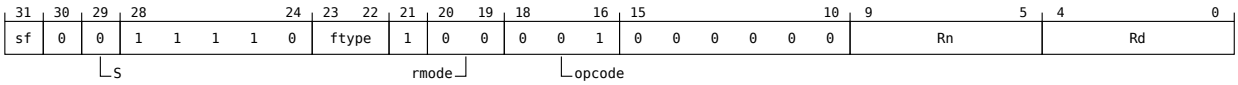
C7.2.76 FCVTNU (scalar)

Floating-point convert to unsigned integer, rounding to nearest with ties to even (scalar)

This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round to Nearest rounding mode, and writes the result to the general-purpose destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Encoding for the Half-precision to 32-bit variant
(FEAT_FP16) Applies when `(sf == 0 && ftype == 11)`

```
FCVTNU <Wd>, <Hn>
```

Encoding for the Half-precision to 64-bit variant
(FEAT_FP16) Applies when `(sf == 1 && ftype == 11)`

```
FCVTNU <Xd>, <Hn>
```

Encoding for the Single-precision to 32-bit variant
(FEAT_FP) Applies when `(sf == 0 && ftype == 00)`

```
FCVTNU <Wd>, <Sn>
```

Encoding for the Single-precision to 64-bit variant
(FEAT_FP) Applies when `(sf == 1 && ftype == 00)`

```
FCVTNU <Xd>, <Sn>
```

Encoding for the Double-precision to 32-bit variant
(FEAT_FP) Applies when `(sf == 0 && ftype == 01)`

```
FCVTNU <Wd>, <Dn>
```

Encoding for the Double-precision to 64-bit variant
(FEAT_FP) Applies when `(sf == 1 && ftype == 01)`

```
FCVTNU <Xd>, <Dn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if ftype == '10' then EndOfDecode(Decode_UNDEF);
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer intsize = 32 << UInt(sf);
constant integer fltsize = 8 << UInt(ftype EOR '10');

constant FPRounding rounding = FPRounding_TIEEVEN;
constant boolean unsigned    = TRUE;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Hn**> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Sn**> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<**Dn**> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(fltsize) fltval = V[n, fltsize];
constant integer fracbits = 0;

X[d, intsize] = FPToFixed(fltval, fracbits, unsigned, FPCR, rounding, intsize);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.77 FCVTPS (vector)

Floating-point convert to signed integer, rounding toward plus infinity (vector)

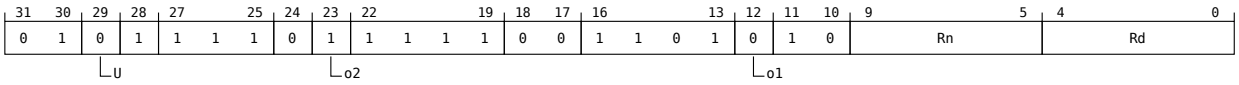
This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision , Scalar single-precision and double-precision , Vector half-precision and Vector single-precision and double-precision

Scalar half-precision
(FEAT_AdvSIMD && FEAT_FP16)



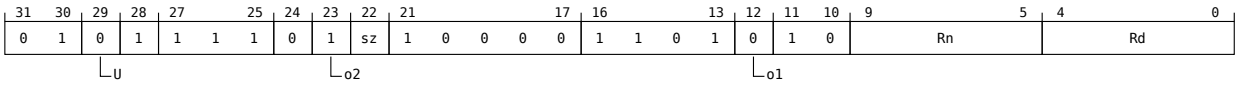
Encoding

```
FCVTPS <Hd>, <Hn>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 16;
constant integer datasize = esize;
constant integer elements = 1;
constant FPRounding rounding = FPDecodeRounding(o1:o2);
constant boolean unsigned = FALSE;
```

Scalar single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

```
FCVTPS <V><d>, <V><n>
```

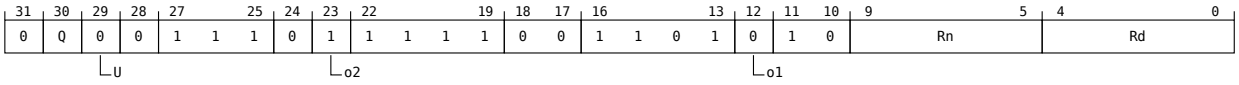
Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 32 << UInt(sz);
constant integer datasize = esize;
constant integer elements = 1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant FPRounding rounding = FPDecodeRounding(o1:o2);
constant boolean unsigned = FALSE;
```

Vector half-precision (FEAT_AdvSIMD && FEAT_FP16)



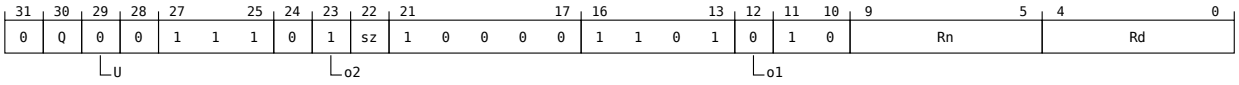
Encoding

```
FCVTPS <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
constant FPRounding rounding = FPDecodeRounding(o1:o2);
constant boolean unsigned = FALSE;
```

Vector single-precision and double-precision (FEAT_AdvSIMD)



Encoding

```
FCVTPS <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
constant FPRounding rounding = FPDecodeRounding(o1:o2);
constant boolean unsigned = FALSE;
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<**V**> Is a width specifier, encoded in “sz”:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	<V>
0	S
1	D

<**d**> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<**n**> Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> For the “Vector half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Vector single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(datasize) operand = V[n, datasize];

bits(datasize) result;
constant integer fracbits = 0;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, fracbits, unsigned, FPCR, rounding, esize);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

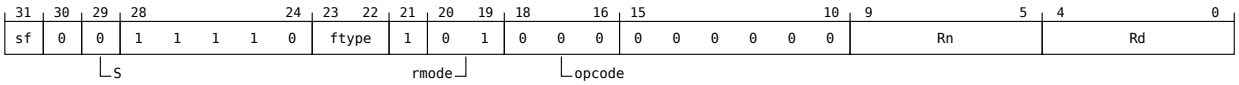
C7.2.78 FCVTPS (scalar)

Floating-point convert to signed integer, rounding toward plus infinity (scalar)

This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round towards Plus Infinity rounding mode, and writes the result to the general-purpose destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Encoding for the Half-precision to 32-bit variant
(FEAT_FP16) Applies when `(sf == 0 && ftype == 11)`

```
FCVTPS <Wd>, <Hn>
```

Encoding for the Half-precision to 64-bit variant
(FEAT_FP16) Applies when `(sf == 1 && ftype == 11)`

```
FCVTPS <Xd>, <Hn>
```

Encoding for the Single-precision to 32-bit variant
(FEAT_FP) Applies when `(sf == 0 && ftype == 00)`

```
FCVTPS <Wd>, <Sn>
```

Encoding for the Single-precision to 64-bit variant
(FEAT_FP) Applies when `(sf == 1 && ftype == 00)`

```
FCVTPS <Xd>, <Sn>
```

Encoding for the Double-precision to 32-bit variant
(FEAT_FP) Applies when `(sf == 0 && ftype == 01)`

```
FCVTPS <Wd>, <Dn>
```

Encoding for the Double-precision to 64-bit variant
(FEAT_FP) Applies when `(sf == 1 && ftype == 01)`

```
FCVTPS <Xd>, <Dn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if ftype == '10' then EndOfDecode(Decode_UNDEF);
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer intsize = 32 << UInt(sf);
constant integer fltsize = 8 << UInt(ftype EOR '10');

constant FPRounding rounding = FPRounding_POSINF;
constant boolean unsigned    = FALSE;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Hn**> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Sn**> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<**Dn**> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(fltsize) fltval = V[n, fltsize];
constant integer fracbits = 0;

X[d, intsize] = FPToFixed(fltval, fracbits, unsigned, FPCR, rounding, intsize);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.79 FCVTPU (vector)

Floating-point convert to unsigned integer, rounding toward plus infinity (vector)

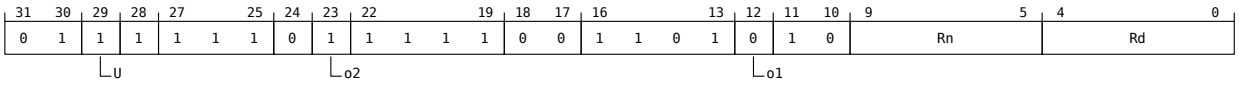
This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision , Scalar single-precision and double-precision , Vector half-precision and Vector single-precision and double-precision

Scalar half-precision
(FEAT_AdvSIMD && FEAT_FP16)



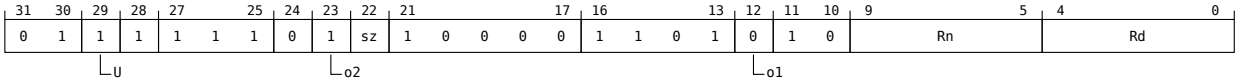
Encoding

```
FCVTPU <Hd>, <Hn>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 16;
constant integer datasize = esize;
constant integer elements = 1;
constant FPRounding rounding = FPDecodeRounding(o1:o2);
constant boolean unsigned = TRUE;
```

Scalar single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

```
FCVTPU <V><d>, <V><n>
```

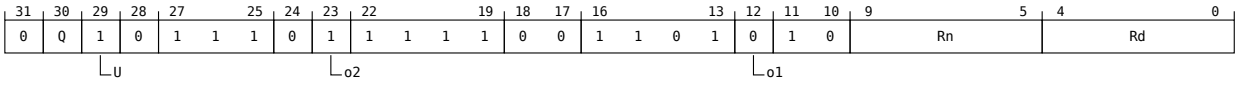
Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 32 << UInt(sz);
constant integer datasize = esize;
constant integer elements = 1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant FPRounding rounding = FPDecodeRounding(o1:o2);
constant boolean unsigned = TRUE;
```

Vector half-precision (FEAT_AdvSIMD && FEAT_FP16)



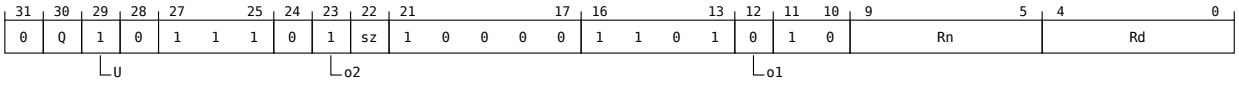
Encoding

```
FCVTPU <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
constant FPRounding rounding = FPDecodeRounding(o1:o2);
constant boolean unsigned = TRUE;
```

Vector single-precision and double-precision (FEAT_AdvSIMD)



Encoding

```
FCVTPU <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
constant FPRounding rounding = FPDecodeRounding(o1:o2);
constant boolean unsigned = TRUE;
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<**V**> Is a width specifier, encoded in “sz”:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	<V>
0	S
1	D

<**d**> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<**n**> Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> For the “Vector half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Vector single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(datasize) operand = V[n, datasize];

bits(datasize) result;
constant integer fracbits = 0;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, fracbits, unsigned, FPCR, rounding, esize);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

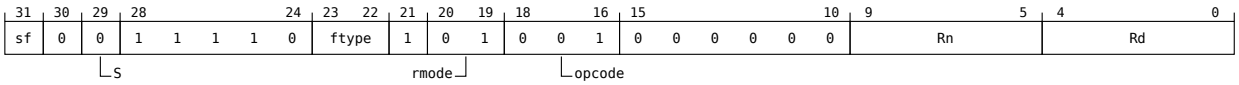
C7.2.80 FCVTPU (scalar)

Floating-point convert to unsigned integer, rounding toward plus infinity (scalar)

This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round towards Plus Infinity rounding mode, and writes the result to the general-purpose destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Encoding for the Half-precision to 32-bit variant
(FEAT_FP16) Applies when `(sf == 0 && ftype == 11)`

```
FCVTPU <Wd>, <Hn>
```

Encoding for the Half-precision to 64-bit variant
(FEAT_FP16) Applies when `(sf == 1 && ftype == 11)`

```
FCVTPU <Xd>, <Hn>
```

Encoding for the Single-precision to 32-bit variant
(FEAT_FP) Applies when `(sf == 0 && ftype == 00)`

```
FCVTPU <Wd>, <Sn>
```

Encoding for the Single-precision to 64-bit variant
(FEAT_FP) Applies when `(sf == 1 && ftype == 00)`

```
FCVTPU <Xd>, <Sn>
```

Encoding for the Double-precision to 32-bit variant
(FEAT_FP) Applies when `(sf == 0 && ftype == 01)`

```
FCVTPU <Wd>, <Dn>
```

Encoding for the Double-precision to 64-bit variant
(FEAT_FP) Applies when `(sf == 1 && ftype == 01)`

```
FCVTPU <Xd>, <Dn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if ftype == '10' then EndOfDecode(Decode_UNDEF);
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer intsize = 32 << UInt(sf);
constant integer fltsize = 8 << UInt(ftype EOR '10');

constant FPRounding rounding = FPRounding_POSINF;
constant boolean unsigned    = TRUE;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Hn**> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Sn**> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<**Dn**> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(fltsize) fltval = V[n, fltsize];
constant integer fracbits = 0;

X[d, intsize] = FPToFixed(fltval, fracbits, unsigned, FPCR, rounding, intsize);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.81 FCVTXN, FCVTXN2

Floating-point convert to lower precision narrow, rounding to odd (vector)

This instruction reads each double-precision element in the source SIMD&FP register, converts each value to single-precision format using the Round to Odd rounding mode, and writes the results to the destination SIMD&FP register.

Note

This instruction uses the Round to Odd rounding mode, which is not defined by the IEEE 754-2008 standard. This rounding mode ensures that if the result of the conversion is inexact, the least significant bit of the mantissa is forced to 1. This rounding mode enables a floating-point value to be converted to a lower precision format via an intermediate precision format while avoiding double rounding errors. For example, a 64-bit floating-point value can be converted to a correctly rounded 16-bit floating-point value by first using this instruction to produce a 32-bit value and then using another instruction with the wanted rounding mode to convert the 32-bit value to the final 16-bit floating-point value.

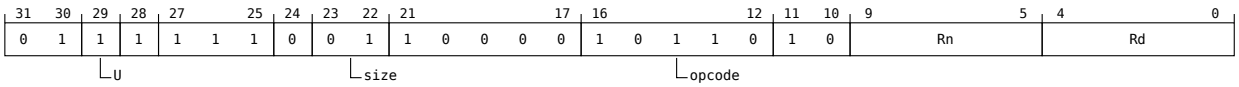
`FCVTXN` writes the vector to the lower half of the destination register and clears the upper half to 0. `FCVTXN2` writes the vector to the upper half of the destination register without affecting the other bits of the register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



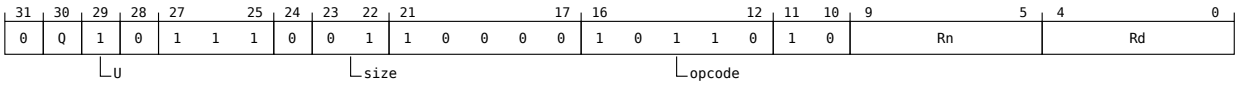
Encoding

```
FCVTXN S<d>, D<n>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 32;
constant integer datasize = esize;
constant integer elements = 1;
constant integer part = 0;
```

Vector
(FEAT_AdvSIMD)



Encoding

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
FCVTXN{2} <Vd>.<Tb>, <Vn>.2D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 32;
constant integer datasize = 64;
constant integer elements = 2;
constant integer part = UInt(Q);
```

Assembler Symbols

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in "Q":

Q	<Tb>
0	2S
1	4S

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(2*datasize) operand = V[n, 2*datasize];
bits(datasize) result;

for e = 0 to elements-1
    Elem[result, e, esize] = FPConvert(Elem[operand, e, 2*esize], FPCR, FPRounding_ODD, esize);

Vpart[d, part, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.82 FCVTZS (vector, fixed-point)

Floating-point convert to signed fixed-point, rounding toward zero (vector)

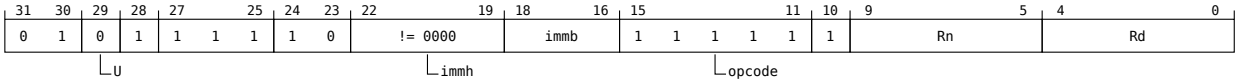
This instruction converts a scalar or each element in a vector from floating-point to fixed-point signed integer using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



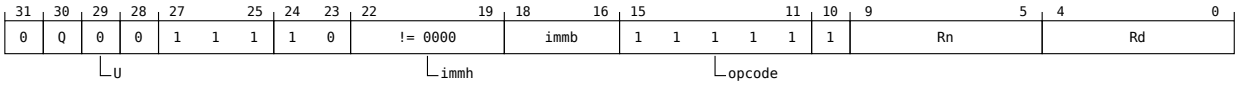
Encoding

```
FCVTZS <V><d>, <V><n>, #<fbits>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh IN {'000x'} || (immh IN {'001x'} && !IsFeatureImplemented(FEAT_FP16)) then
    EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = if immh IN {'1xxx'} then 64 else if immh IN {'01xx'} then 32 else 16;
constant integer datasize = esize;
constant integer elements = 1;

constant integer fracbits = (esize * 2) - UInt(immh:immb);
constant boolean unsigned = FALSE;
constant FPRounding rounding = FPRounding_ZERO;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
FCVTZS <Vd>.<T>, <Vn>.<T>, #<fbits>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh IN {'000x'} || (immh IN {'001x'} && !IsFeatureImplemented(FEAT_FP16)) then
    EndOfDecode(Decode_UNDEF);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if immh<3>:Q == '10' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = if immh IN {'1xxx'} then 64 else if immh IN {'01xx'} then 32 else 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;

constant integer fracbits = (esize * 2) - UInt(immh:immb);
constant boolean unsigned = FALSE;
constant FPRounding rounding = FPRounding_ZERO;
```

Assembler Symbols

<**V**> Is a width specifier, encoded in "immh":

immh	<V>	Architectural Feature
0001	RESERVED	-
001x	H	FEAT_FP16
01xx	S	-
1xxx	D	-

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**n**> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<**fbits**> For the "Scalar" variant: is the number of fractional bits, in the range 1 to the operand width, encoded in "immh:immb":

immh	<fbits>
0001	RESERVED
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	128 - UInt(immh:immb)

For the "Vector" variant: is the number of fractional bits, in the range 1 to the element width, encoded in "immh:immb":

immh	<fbits>
0001	RESERVED
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	128 - UInt(immh:immb)

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> Is an arrangement specifier, encoded in "immh:Q":

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

immh	Q	<T>	Architectural Feature
0001	x	RESERVED	-
001x	0	4H	FEAT_FP16
001x	1	8H	FEAT_FP16
01xx	0	2S	-
01xx	1	4S	-
1xxx	0	RESERVED	-
1xxx	1	2D	-

<**Vn**> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand  = V[n, datasize];

bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, fracbits, unsigned, FPCR, rounding, esize);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.83 FCVTZS (vector, integer)

Floating-point convert to signed integer, rounding toward zero (vector)

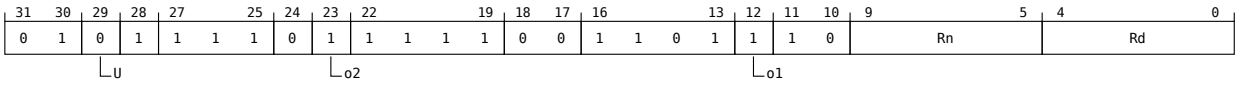
This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision , Scalar single-precision and double-precision , Vector half-precision and Vector single-precision and double-precision

Scalar half-precision
(FEAT_AdvSIMD && FEAT_FP16)



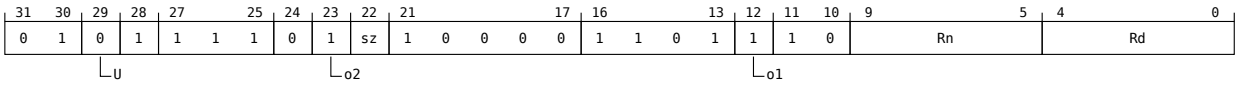
Encoding

```
FCVTZS <Hd>, <Hn>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 16;
constant integer datasize = esize;
constant integer elements = 1;
constant FPRounding rounding = FPDecodeRounding(o1:o2);
constant boolean unsigned = FALSE;
```

Scalar single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

```
FCVTZS <V><d>, <V><n>
```

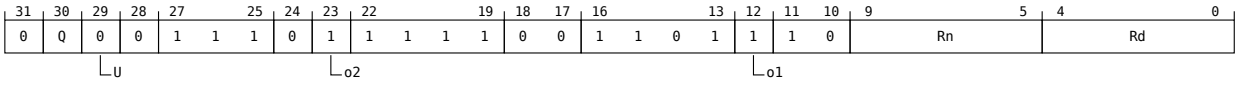
Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 32 << UInt(sz);
constant integer datasize = esize;
constant integer elements = 1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant FPRounding rounding = FPDecodeRounding(o1:o2);
constant boolean unsigned = FALSE;
```

Vector half-precision (FEAT_AdvSIMD && FEAT_FP16)



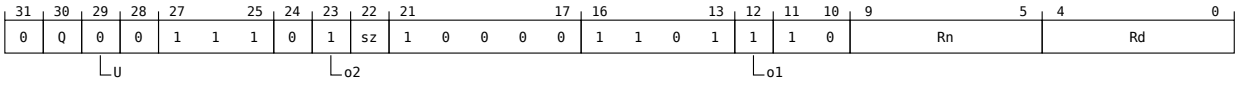
Encoding

```
FCVTZS <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
constant FPRounding rounding = FPDecodeRounding(o1:o2);
constant boolean unsigned = FALSE;
```

Vector single-precision and double-precision (FEAT_AdvSIMD)



Encoding

```
FCVTZS <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
constant FPRounding rounding = FPDecodeRounding(o1:o2);
constant boolean unsigned = FALSE;
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<**V**> Is a width specifier, encoded in “sz”:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	<V>
0	S
1	D

<**d**> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<**n**> Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> For the “Vector half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Vector single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(datasize) operand = V[n, datasize];

bits(datasize) result;
constant integer fracbits = 0;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, fracbits, unsigned, FPCR, rounding, esize);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

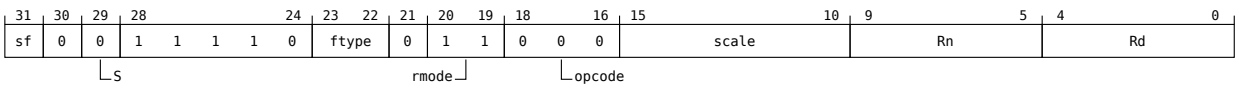
C7.2.84 FCVTZS (scalar, fixed-point)

Floating-point convert to signed fixed-point, rounding toward zero (scalar)

This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit fixed-point signed integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Encoding for the Half-precision to 32-bit variant
(FEAT_FP16) Applies when `(sf == 0 && ftype == 11)`

```
FCVTZS <Wd>, <Hn>, #<fbits>
```

Encoding for the Half-precision to 64-bit variant
(FEAT_FP16) Applies when `(sf == 1 && ftype == 11)`

```
FCVTZS <Xd>, <Hn>, #<fbits>
```

Encoding for the Single-precision to 32-bit variant
(FEAT_FP) Applies when `(sf == 0 && ftype == 00)`

```
FCVTZS <Wd>, <Sn>, #<fbits>
```

Encoding for the Single-precision to 64-bit variant
(FEAT_FP) Applies when `(sf == 1 && ftype == 00)`

```
FCVTZS <Xd>, <Sn>, #<fbits>
```

Encoding for the Double-precision to 32-bit variant
(FEAT_FP) Applies when `(sf == 0 && ftype == 01)`

```
FCVTZS <Wd>, <Dn>, #<fbits>
```

Encoding for the Double-precision to 64-bit variant
(FEAT_FP) Applies when `(sf == 1 && ftype == 01)`

```
FCVTZS <Xd>, <Dn>, #<fbits>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if ftype == '10' then EndOfDecode(Decode_UNDEF);
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);
if sf == '0' && scale<5> == '0' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer intsize = 32 << UInt(sf);
constant integer decode_fltsize = 8 << UInt(ftype EOR '10');
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer fracbits = 64 - UInt(scale);

constant FPRounding rounding = FPRounding_ZERO;
constant boolean unsigned = FALSE;
```

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Hn**> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<**fbits**> For the "Double-precision to 32-bit", "Half-precision to 32-bit", and "Single-precision to 32-bit" variants: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 32, encoded as 64 minus "scale".

For the "Double-precision to 64-bit", "Half-precision to 64-bit", and "Single-precision to 64-bit" variants: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 64, encoded as 64 minus "scale".

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Sn**> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<**Dn**> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(decode_fltsize) fltval = V[n, decode_fltsize];
X[d, intsize] = FPToFixed(fltval, fracbits, unsigned, FPCR, rounding, intsize);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

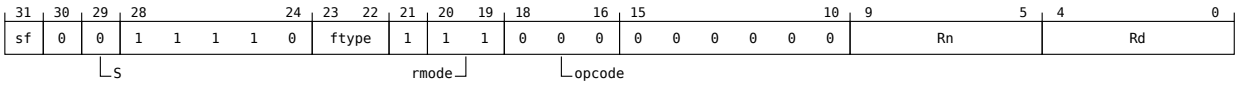
C7.2.85 FCVTZS (scalar, integer)

Floating-point convert to signed integer, rounding toward zero (scalar)

This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Encoding for the Half-precision to 32-bit variant
(FEAT_FP16) Applies when `(sf == 0 && ftype == 11)`

```
FCVTZS <Wd>, <Hn>
```

Encoding for the Half-precision to 64-bit variant
(FEAT_FP16) Applies when `(sf == 1 && ftype == 11)`

```
FCVTZS <Xd>, <Hn>
```

Encoding for the Single-precision to 32-bit variant
(FEAT_FP) Applies when `(sf == 0 && ftype == 00)`

```
FCVTZS <Wd>, <Sn>
```

Encoding for the Single-precision to 64-bit variant
(FEAT_FP) Applies when `(sf == 1 && ftype == 00)`

```
FCVTZS <Xd>, <Sn>
```

Encoding for the Double-precision to 32-bit variant
(FEAT_FP) Applies when `(sf == 0 && ftype == 01)`

```
FCVTZS <Wd>, <Dn>
```

Encoding for the Double-precision to 64-bit variant
(FEAT_FP) Applies when `(sf == 1 && ftype == 01)`

```
FCVTZS <Xd>, <Dn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if ftype == '10' then EndOfDecode(Decode_UNDEF);
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer intsize = 32 << UInt(sf);
constant integer fltsize = 8 << UInt(ftype EOR '10');

constant FPRounding rounding = FPRounding_ZERO;
constant boolean unsigned    = FALSE;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Hn**> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Sn**> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<**Dn**> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(fltsize) fltval = V[n, fltsize];
constant integer fracbits = 0;

X[d, intsize] = FPToFixed(fltval, fracbits, unsigned, FPCR, rounding, intsize);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.86 FCVTZU (vector, fixed-point)

Floating-point convert to unsigned fixed-point, rounding toward zero (vector)

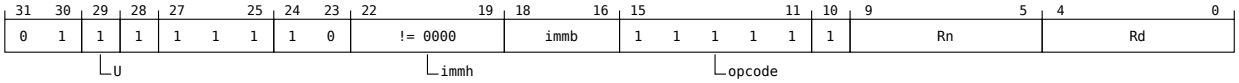
This instruction converts a scalar or each element in a vector from floating-point to fixed-point unsigned integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



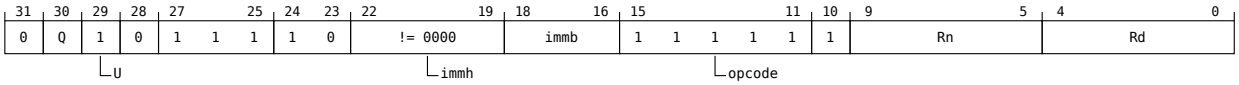
Encoding

```
FCVTZU <V><d>, <V><n>, #<fbits>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh IN {'000x'} || (immh IN {'001x'} && !IsFeatureImplemented(FEAT_FP16)) then
    EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = if immh IN {'1xxx'} then 64 else if immh IN {'01xx'} then 32 else 16;
constant integer datasize = esize;
constant integer elements = 1;

constant integer fracbits = (esize * 2) - UInt(immh:immb);
constant boolean unsigned = TRUE;
constant FPRounding rounding = FPRounding_ZERO;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
FCVTZU <Vd>.<T>, <Vn>.<T>, #<fbits>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh IN {'000x'} || (immh IN {'001x'} && !IsFeatureImplemented(FEAT_FP16)) then
    EndOfDecode(Decode_UNDEF);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if immh<3>:Q == '10' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = if immh IN {'1xxx'} then 64 else if immh IN {'01xx'} then 32 else 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;

constant integer fracbits = (esize * 2) - UInt(immh:immb);
constant boolean unsigned = TRUE;
constant FPRounding rounding = FPRounding_ZERO;
```

Assembler Symbols

<**V**> Is a width specifier, encoded in "immh":

immh	<V>	Architectural Feature
0001	RESERVED	-
001x	H	FEAT_FP16
01xx	S	-
1xxx	D	-

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**n**> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<**fbits**> For the "Scalar" variant: is the number of fractional bits, in the range 1 to the operand width, encoded in "immh:immb":

immh	<fbits>
0001	RESERVED
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	128 - UInt(immh:immb)

For the "Vector" variant: is the number of fractional bits, in the range 1 to the element width, encoded in "immh:immb":

immh	<fbits>
0001	RESERVED
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	128 - UInt(immh:immb)

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> Is an arrangement specifier, encoded in "immh:Q":

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

immh	Q	<T>	Architectural Feature
0001	x	RESERVED	-
001x	0	4H	FEAT_FP16
001x	1	8H	FEAT_FP16
01xx	0	2S	-
01xx	1	4S	-
1xxx	0	RESERVED	-
1xxx	1	2D	-

<**Vn**> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand  = V[n, datasize];

bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, fracbits, unsigned, FPCR, rounding, esize);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.87 FCVTZU (vector, integer)

Floating-point convert to unsigned integer, rounding toward zero (vector)

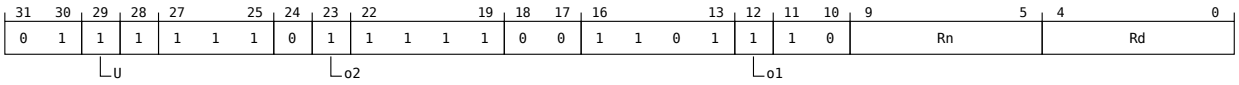
This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision , Scalar single-precision and double-precision , Vector half-precision and Vector single-precision and double-precision

Scalar half-precision
(FEAT_AdvSIMD && FEAT_FP16)



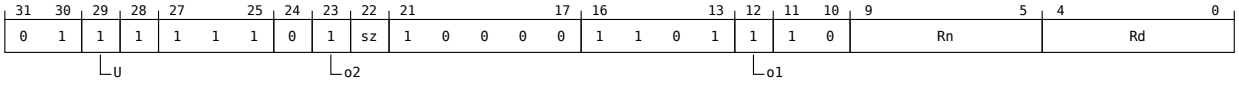
Encoding

```
FCVTZU <Hd>, <Hn>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 16;
constant integer datasize = esize;
constant integer elements = 1;
constant FPRounding rounding = FPDecodeRounding(o1:o2);
constant boolean unsigned = TRUE;
```

Scalar single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

```
FCVTZU <V><d>, <V><n>
```

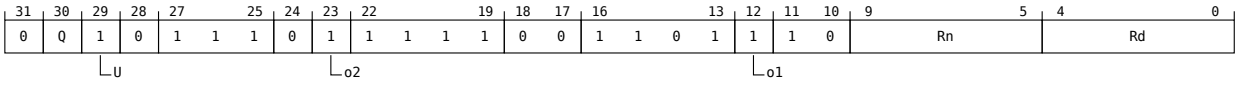
Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 32 << UInt(sz);
constant integer datasize = esize;
constant integer elements = 1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant FPRounding rounding = FPDecodeRounding(o1:o2);
constant boolean unsigned = TRUE;
```

Vector half-precision (FEAT_AdvSIMD && FEAT_FP16)



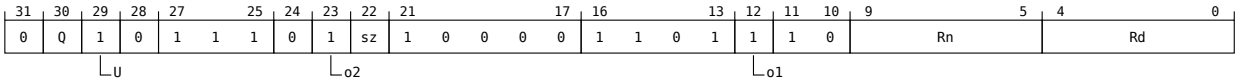
Encoding

```
FCVTZU <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
constant FPRounding rounding = FPDecodeRounding(o1:o2);
constant boolean unsigned = TRUE;
```

Vector single-precision and double-precision (FEAT_AdvSIMD)



Encoding

```
FCVTZU <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
constant FPRounding rounding = FPDecodeRounding(o1:o2);
constant boolean unsigned = TRUE;
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<**V**> Is a width specifier, encoded in “sz”:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	<V>
0	S
1	D

<**d**> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<**n**> Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> For the “Vector half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Vector single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(datasize) operand = V[n, datasize];

bits(datasize) result;
constant integer fracbits = 0;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, fracbits, unsigned, FPCR, rounding, esize);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

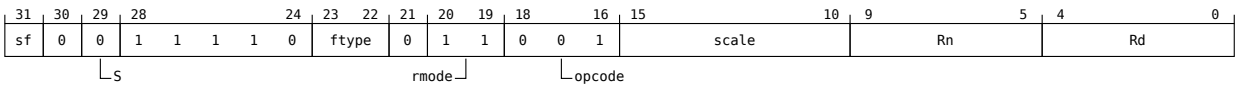
C7.2.88 FCVTZU (scalar, fixed-point)

Floating-point convert to unsigned fixed-point, rounding toward zero (scalar)

This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit fixed-point unsigned integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Encoding for the Half-precision to 32-bit variant
(FEAT_FP16) Applies when `(sf == 0 && ftype == 11)`

```
FCVTZU <Wd>, <Hn>, #<fbits>
```

Encoding for the Half-precision to 64-bit variant
(FEAT_FP16) Applies when `(sf == 1 && ftype == 11)`

```
FCVTZU <Xd>, <Hn>, #<fbits>
```

Encoding for the Single-precision to 32-bit variant
(FEAT_FP) Applies when `(sf == 0 && ftype == 00)`

```
FCVTZU <Wd>, <Sn>, #<fbits>
```

Encoding for the Single-precision to 64-bit variant
(FEAT_FP) Applies when `(sf == 1 && ftype == 00)`

```
FCVTZU <Xd>, <Sn>, #<fbits>
```

Encoding for the Double-precision to 32-bit variant
(FEAT_FP) Applies when `(sf == 0 && ftype == 01)`

```
FCVTZU <Wd>, <Dn>, #<fbits>
```

Encoding for the Double-precision to 64-bit variant
(FEAT_FP) Applies when `(sf == 1 && ftype == 01)`

```
FCVTZU <Xd>, <Dn>, #<fbits>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if ftype == '10' then EndOfDecode(Decode_UNDEF);
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);
if sf == '0' && scale<5> == '0' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer intsize = 32 << UInt(sf);
constant integer decode_fltsize = 8 << UInt(ftype EOR '10');
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer fracbits = 64 - UInt(scale);

constant FPRounding rounding = FPRounding_ZERO;
constant boolean unsigned = TRUE;
```

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Hn**> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<**fbits**> For the "Double-precision to 32-bit", "Half-precision to 32-bit", and "Single-precision to 32-bit" variants: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 32, encoded as 64 minus "scale".

For the "Double-precision to 64-bit", "Half-precision to 64-bit", and "Single-precision to 64-bit" variants: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 64, encoded as 64 minus "scale".

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Sn**> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<**Dn**> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(decode_fltsize) fltval = V[n, decode_fltsize];
X[d, intsize] = FPToFixed(fltval, fracbits, unsigned, FPCR, rounding, intsize);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

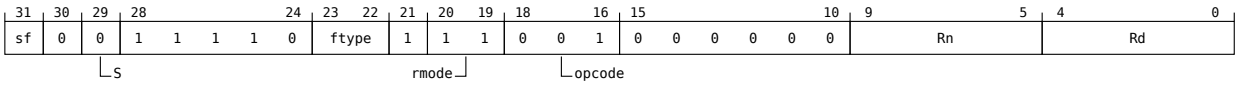
C7.2.89 FCVTZU (scalar, integer)

Floating-point convert to unsigned integer, rounding toward zero (scalar)

This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Encoding for the Half-precision to 32-bit variant
(FEAT_FP16) Applies when `(sf == 0 && ftype == 11)`

```
FCVTZU <Wd>, <Hn>
```

Encoding for the Half-precision to 64-bit variant
(FEAT_FP16) Applies when `(sf == 1 && ftype == 11)`

```
FCVTZU <Xd>, <Hn>
```

Encoding for the Single-precision to 32-bit variant
(FEAT_FP) Applies when `(sf == 0 && ftype == 00)`

```
FCVTZU <Wd>, <Sn>
```

Encoding for the Single-precision to 64-bit variant
(FEAT_FP) Applies when `(sf == 1 && ftype == 00)`

```
FCVTZU <Xd>, <Sn>
```

Encoding for the Double-precision to 32-bit variant
(FEAT_FP) Applies when `(sf == 0 && ftype == 01)`

```
FCVTZU <Wd>, <Dn>
```

Encoding for the Double-precision to 64-bit variant
(FEAT_FP) Applies when `(sf == 1 && ftype == 01)`

```
FCVTZU <Xd>, <Dn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if ftype == '10' then EndOfDecode(Decode_UNDEF);
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer intsize = 32 << UInt(sf);
constant integer fltsize = 8 << UInt(ftype EOR '10');

constant FPRounding rounding = FPRounding_ZERO;
constant boolean unsigned    = TRUE;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Hn**> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<**Sn**> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<**Dn**> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(fltsize) fltval = V[n, fltsize];
constant integer fracbits = 0;

X[d, intsize] = FPToFixed(fltval, fracbits, unsigned, FPCR, rounding, intsize);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.90 FDIV (vector)

Floating-point divide (vector)

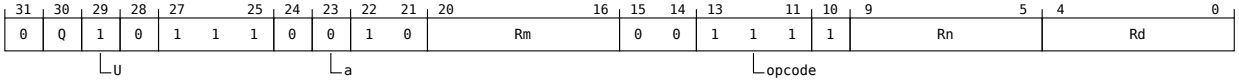
This instruction divides the floating-point values in the elements in the first source SIMD&FP register, by the floating-point values in the corresponding elements in the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(FEAT_AdvSIMD && FEAT_FP16)



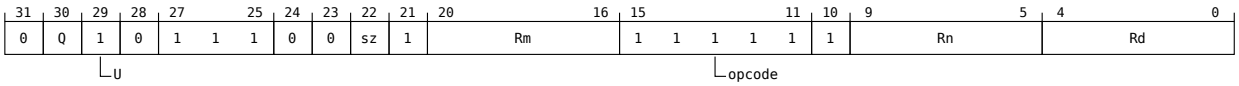
Encoding

```
FDIV <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

```
FDIV <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> For the "Half-precision" variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the "Single-precision and double-precision" variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPDiv(element1, element2, FPCR);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

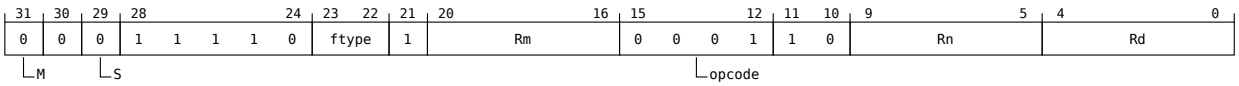
C7.2.91 FDIV (scalar)

Floating-point divide (scalar)

This instruction divides the floating-point value of the first source SIMD&FP register by the floating-point value of the second source SIMD&FP register, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Encoding for the Half-precision variant
(FEAT_FP16) Applies when `(ftype == 11)`

```
FDIV <Hd>, <Hn>, <Hm>
```

Encoding for the Single-precision variant
(FEAT_FP) Applies when `(ftype == 00)`

```
FDIV <Sd>, <Sn>, <Sm>
```

Encoding for the Double-precision variant
(FEAT_FP) Applies when `(ftype == 01)`

```
FDIV <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if ftype == '10' then EndOfDecode(Decode_UNDEF);
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 8 << UInt(ftype EOR '10');
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Hm**> Is the 16-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<**Sd**> Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Sn**> Is the 32-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Sm**> Is the 32-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<**Dd**> Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Dn**> Is the 64-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**Dm**> Is the 64-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(esize) operand1 = V[n, esize];
constant bits(esize) operand2 = V[m, esize];

V[d, esize] = FPDiv(operand1, operand2, FPCR);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.92 FJCVTZS

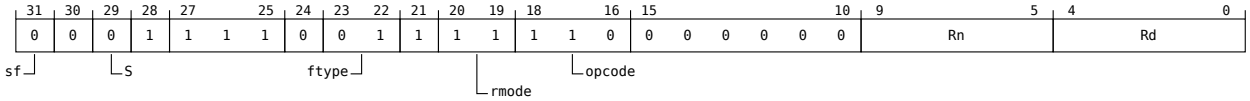
Floating-point Javascript convert to signed fixed-point, rounding toward zero

This instruction converts the double-precision floating-point value in the SIMD&FP source register to a 32-bit signed integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register. If the result is too large to be represented as a signed 32-bit integer, then the result is the integer modulo 2^{32}, as held in a 32-bit signed integer.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Double-precision to 32-bit
(FEAT_JSCVT)



Encoding

```
FJCVTZS <Wd>, <Dn>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_JSCVT) then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
```

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(64) fltval = V[n, 64];

bits(32) intval;
bit z;
(intval, z) = FPToFixedJS(fltval, FPCR);

X[d, 32] = intval;
PSTATE.<N,Z,C,V> = '0':z:'00';
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

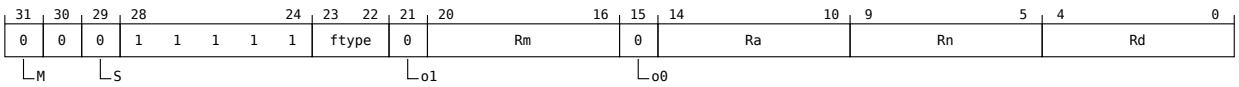
C7.2.93 FMADD

Floating-point fused multiply-add (scalar)

This instruction multiplies the values of the first two SIMD&FP source registers, adds the product to the value of the third SIMD&FP source register, and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Encoding for the Half-precision variant
(FEAT_FP16) Applies when `(ftype == 11)`

```
FMADD <Hd>, <Hn>, <Hm>, <Ha>
```

Encoding for the Single-precision variant
(FEAT_FP) Applies when `(ftype == 00)`

```
FMADD <Sd>, <Sn>, <Sm>, <Sa>
```

Encoding for the Double-precision variant
(FEAT_FP) Applies when `(ftype == 01)`

```
FMADD <Dd>, <Dn>, <Dm>, <Da>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if ftype == '10' then EndOfDecode(Decode_UNDEF);
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer a = UInt(Ra);

constant integer esize = 8 << UInt(ftype EOR '10');
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the “Rn” field.

<**Hm**> Is the 16-bit name of the second SIMD&FP source register holding the multiplier, encoded in the “Rm” field.

<**Ha**> Is the 16-bit name of the third SIMD&FP source register holding the addend, encoded in the “Ra” field.

<**Sd**> Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Sn**> Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the “Rn” field.

<**Sm**> Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the “Rm” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**Sa**> Is the 32-bit name of the third SIMD&FP source register holding the addend, encoded in the “Ra” field.

<**Dd**> Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Dn**> Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the “Rn” field.

<**Dm**> Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the “Rm” field.

<**Da**> Is the 64-bit name of the third SIMD&FP source register holding the addend, encoded in the “Ra” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(esize) addend   = V[a, esize];
constant bits(esize) operand1 = V[n, esize];
constant bits(esize) operand2 = V[m, esize];

V[d, esize] = FPMulAdd(addend, operand1, operand2, FPCR);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.94 FMAX (vector)

Floating-point maximum (vector)

This instruction compares corresponding vector elements in the two source SIMD&FP registers, places the larger of each of the two floating-point values into a vector, and writes the vector to the destination SIMD&FP register.

When FPCR.AH is 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows:

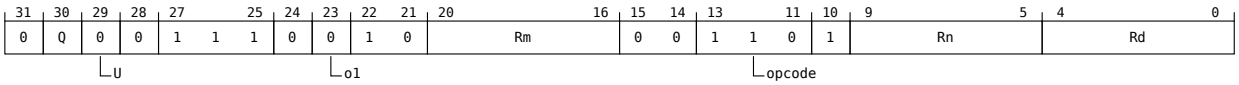
- If both elements are zeros, regardless of the sign of either zero, the result is the second element.
- If either element is a NaN, regardless of the value of FPCR.DN, the result is the second element.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(FEAT_AdvSIMD && FEAT_FP16)



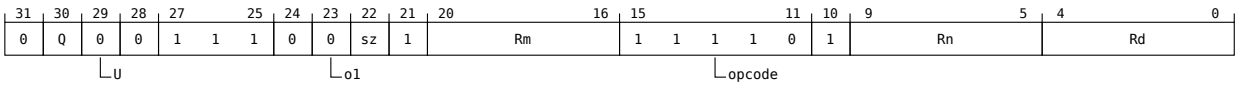
Encoding

```
FMAX <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
FMAX <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the "Half-precision" variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the "Single-precision and double-precision" variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPMax(element1, element2, FPCR);
V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.95 FMAX (scalar)

Floating-point maximum (scalar)

This instruction compares the two source SIMD&FP registers, and writes the larger of the two floating-point values to the destination SIMD&FP register.

When FPCR.AH is 0, the behavior is as follows:

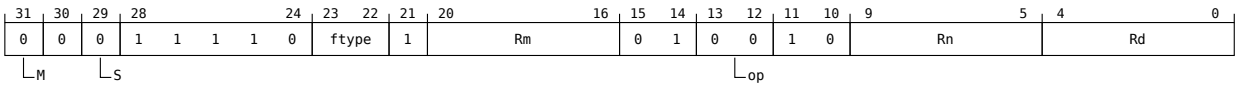
- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either value is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either value is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows:

- If both values are zeros, regardless of the sign of either zero, the result is the second value.
- If either value is a NaN, regardless of the value of FPCR.DN, the result is the second value.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Encoding for the Half-precision variant
(FEAT_FP16) Applies when `(ftype == 11)`

```
FMAX <Hd>, <Hn>, <Hm>
```

Encoding for the Single-precision variant
(FEAT_FP) Applies when `(ftype == 00)`

```
FMAX <Sd>, <Sn>, <Sm>
```

Encoding for the Double-precision variant
(FEAT_FP) Applies when `(ftype == 01)`

```
FMAX <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if ftype == '10' then EndOfDecode(Decode_UNDEF);
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 8 << UInt(ftype EOR '10');
```

Assembler Symbols

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Hm**> Is the 16-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<**Sd**> Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Sn**> Is the 32-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Sm**> Is the 32-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<**Dd**> Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Dn**> Is the 64-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Dm**> Is the 64-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(esize) operand1 = V[n, esize];
constant bits(esize) operand2 = V[m, esize];

V[d, esize] = FPMax(operand1, operand2, FPCR);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.96 FMAXNM (vector)

Floating-point maximum number (vector)

This instruction compares corresponding vector elements in the two source SIMD&FP registers, writes the larger of the two floating-point values into a vector, and writes the vector to the destination SIMD&FP register.

Regardless of the value of FPCR.AH, the behavior is as follows:

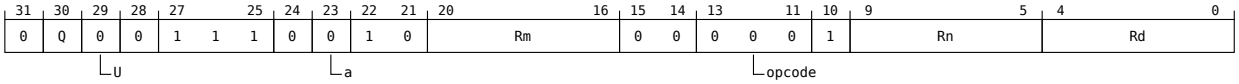
- Negative zero compares less than positive zero.
- If one element is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either element is a signaling NaN or if both elements are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a signaling NaN or if both elements are NaNs, the result is Default NaN.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(FEAT_AdvSIMD && FEAT_FP16)



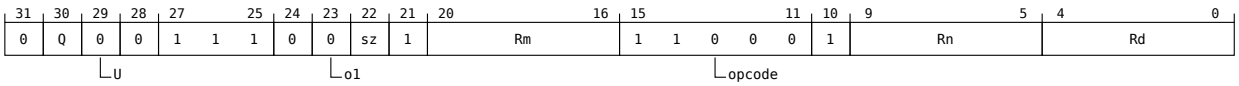
Encoding

```
FMAXNM <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

```
FMAXNM <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> For the "Half-precision" variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the "Single-precision and double-precision" variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPMaxNum(element1, element2, FPCR);
V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.97 FMAXNM (scalar)

Floating-point maximum number (scalar)

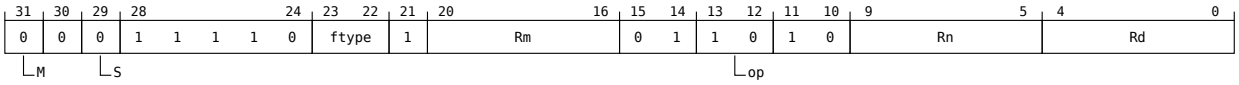
This instruction compares the first and second source SIMD&FP register values, and writes the larger of the two floating-point values to the destination SIMD&FP register.

Regardless of the value of FPCR.AH, the behavior is as follows:

- Negative zero compares less than positive zero.
- If one value is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either value is a signaling NaN or if both values are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either value is a signaling NaN or if both values are NaNs, the result is Default NaN.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Encoding for the Half-precision variant
(FEAT_FP16) Applies when `(ftype == 11)`

```
FMAXNM <Hd>, <Hn>, <Hm>
```

Encoding for the Single-precision variant
(FEAT_FP) Applies when `(ftype == 00)`

```
FMAXNM <Sd>, <Sn>, <Sm>
```

Encoding for the Double-precision variant
(FEAT_FP) Applies when `(ftype == 01)`

```
FMAXNM <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if ftype == '10' then EndOfDecode(Decode_UNDEF);
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 8 << UInt(ftype EOR '10');
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Hm**> Is the 16-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**Sd**> Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Sn**> Is the 32-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Sm**> Is the 32-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<**Dd**> Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Dn**> Is the 64-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Dm**> Is the 64-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(esize) operand1 = V[n, esize];
constant bits(esize) operand2 = V[m, esize];

V[d, esize] = FPMaxNum(operand1, operand2, FPCR);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.98 FMAXNMP (scalar)

Floating-point maximum number of pair of elements (scalar)

This instruction compares two vector elements in the source SIMD&FP register and writes the largest of the floating-point values as a scalar to the destination SIMD&FP register.

Regardless of the value of FPCR.AH, the behavior is as follows for each pairwise operation:

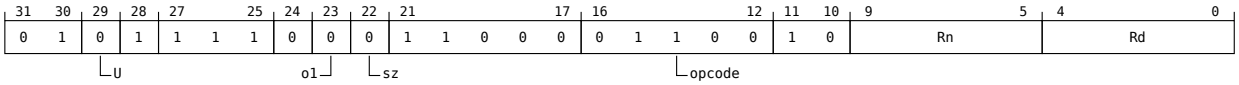
- Negative zero compares less than positive zero.
- If one element is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either element is a signaling NaN or if both elements are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a signaling NaN or if both elements are NaNs, the result is Default NaN.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(FEAT_AdvSIMD && FEAT_FP16)



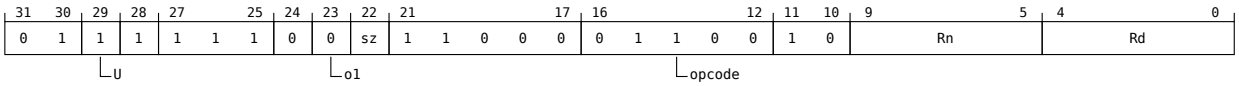
Encoding

```
FMAXNMP H<d>, <Vn>.2H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
if sz == '1' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 16;
constant integer datasize = 32;
```

Single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

```
FMAXNMP <V><d>, <Vn>.<T>
```

Decode for this encoding

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 32 << UInt(sz);
constant integer datasize = esize * 2;
```

Assembler Symbols

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is the destination width specifier, encoded in "sz":

sz	<V>
0	S
1	D

<T> Is the source arrangement specifier, encoded in "sz":

sz	<T>
0	2S
1	2D

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
constant bits(esize) element1 = Elem[operand, 0, esize];
constant bits(esize) element2 = Elem[operand, 1, esize];
V[d, esize] = FPMaxNum(element1, element2, FPCR);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.99 FMAXNMP (vector)

Floating-point maximum number pairwise (vector)

This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the largest of each pair of values into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

Regardless of the value of FPCR.AH, the behavior is as follows for each pairwise operation:

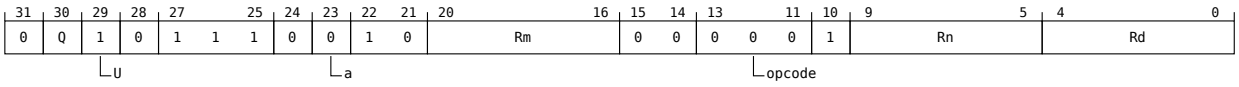
- Negative zero compares less than positive zero.
- If one element is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either element is a signaling NaN or if both elements are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a signaling NaN or if both elements are NaNs, the result is Default NaN.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(FEAT_AdvSIMD && FEAT_FP16)



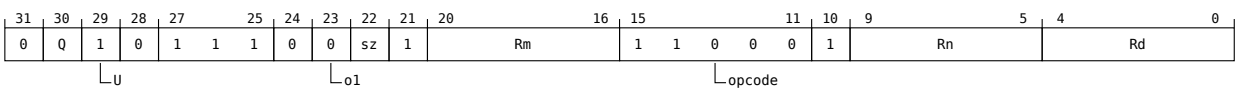
Encoding

```
FMAXNMP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
FMAXNMP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the "Half-precision" variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the "Single-precision and double-precision" variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
constant bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[concat, 2*e, esize];
    element2 = Elem[concat, (2*e)+1, esize];
    Elem[result, e, esize] = FPMaxNum(element1, element2, FPCR);
V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.100 FMAXNMV

Floating-point maximum number across vector

This instruction compares all the vector elements in the source SIMD&FP register, and writes the largest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are floating-point values.

Regardless of the value of FPCR.AH, the behavior is as follows:

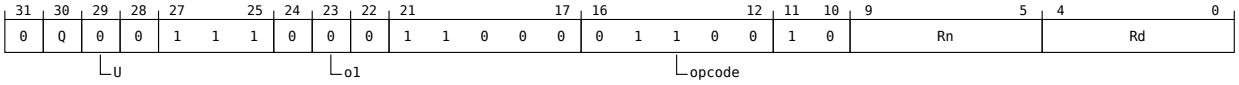
- Negative zero compares less than positive zero.
- If one value is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either value is a signaling NaN or if both values are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either value is a signaling NaN or if both values are NaNs, the result is Default NaN.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision

Half-precision
(FEAT_AdvSIMD && FEAT_FP16)



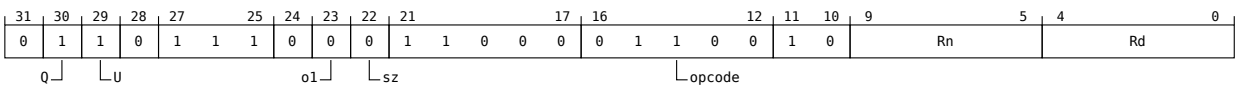
Encoding

```
FMAXNMV <V><d>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
```

Single-precision
(FEAT_AdvSIMD)



Encoding

```
FMAXNMV S<d>, <Vn>.4S
```

Decode for this encoding

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q != '01' then EndOfDecode(Decode_UNDEF);         // .4S only

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 32;
constant integer datasize = 64 << UInt(Q);
```

Assembler Symbols

<**V**> Is the destination width specifier, H.

<**d**> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<**Vn**> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<**T**> Is an arrangement specifier, encoded in “Q”:

Q	<**T**>
0	4H
1	8H

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
V[d, esize] = FPReduce(ReduceOp_FMAXNUM, operand, esize, FPCR);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.101 FMAXP (scalar)

Floating-point maximum of pair of elements (scalar)

This instruction compares two vector elements in the source SIMD&FP register and writes the largest of the floating-point values as a scalar to the destination SIMD&FP register.

When FPCR.AH is 0, the behavior is as follows for each pairwise operation:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows for each pairwise operation:

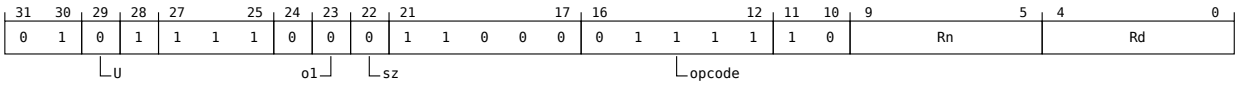
- If both elements are zeros, regardless of the sign of either zero, the result is the second element.
- If either element is a NaN, regardless of the value of FPCR.DN, the result is the second element.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(FEAT_AdvSIMD && FEAT_FP16)



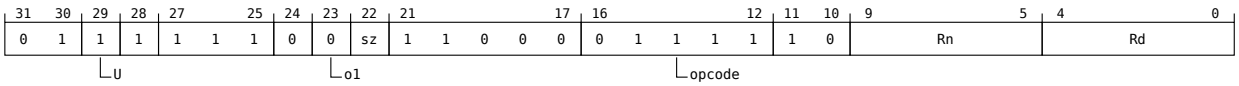
Encoding

```
FMAXP H<d>, <Vn>.2H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
if sz == '1' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 16;
constant integer datasize = 32;
```

Single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
FMAXP <V><d>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 32 << UInt(sz);
constant integer datasize = esize * 2;
```

Assembler Symbols

<**d**> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<**Vn**> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<**V**> Is the destination width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<**T**> Is the source arrangement specifier, encoded in “sz”:

sz	<T>
0	2S
1	2D

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
constant bits(esize) element1 = Elem[operand, 0, esize];
constant bits(esize) element2 = Elem[operand, 1, esize];
V[d, esize] = FPMax(element1, element2, FPCR);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.102 FMAXP (vector)

Floating-point maximum pairwise (vector)

This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements from the concatenated vector, writes the larger of each pair of values into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

When FPCR.AH is 0, the behavior is as follows for each pairwise operation:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows for each pairwise operation:

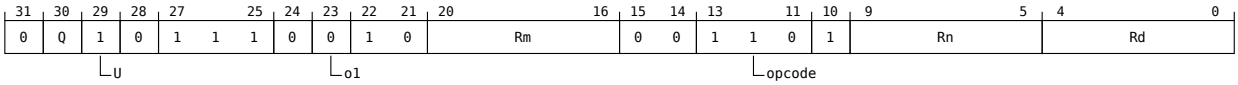
- If both elements are zeros, regardless of the sign of either zero, the result is the second element.
- If either element is a NaN, regardless of the value of FPCR.DN, the result is the second element.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(FEAT_AdvSIMD && FEAT_FP16)



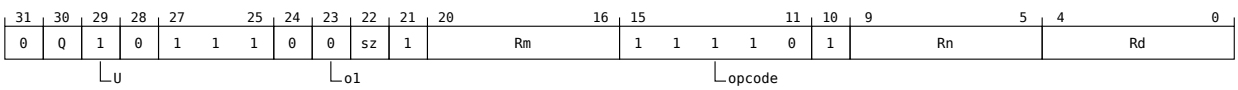
Encoding

```
FMAXP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Single-precision and double-precision
(FEAT_AdvSIMD)



Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding

```
FMAXP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the "Half-precision" variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the "Single-precision and double-precision" variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
constant bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[concat, 2*e, esize];
    element2 = Elem[concat, (2*e)+1, esize];
    Elem[result, e, esize] = FPMax(element1, element2, FPCR);
V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.103 FMAXV

Floating-point maximum across vector

This instruction compares all the vector elements in the source SIMD&FP register, and writes the largest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are floating-point values.

When FPCR.AH is 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either value is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either value is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows:

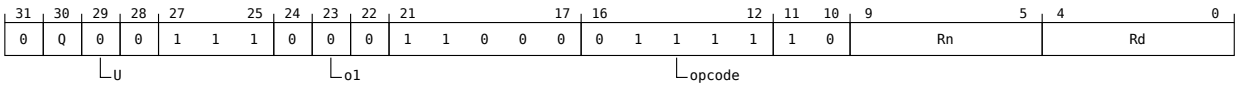
- If both values are zeros, regardless of the sign of either zero, the result is the second value.
- If either value is a NaN, regardless of the value of FPCR.DN, the result is the second value.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision

Half-precision
(FEAT_AdvSIMD && FEAT_FP16)



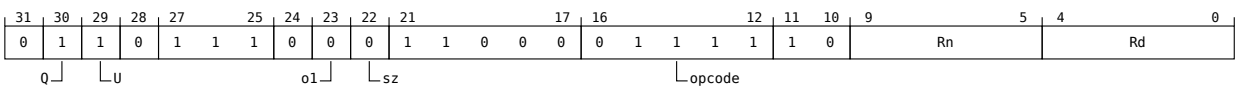
Encoding

```
FMAXV <V><d>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
```

Single-precision
(FEAT_AdvSIMD)



Encoding

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
FMAXV S<d>, <Vn>.4S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q != '01' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 32;
constant integer datasize = 64 << UInt(Q);
```

Assembler Symbols

<V> Is the destination width specifier, H.

<d> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<T> Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
V[d, esize] = FPReduce(ReduceOp_FMAX, operand, esize, FPCR);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.104 FMIN (vector)

Floating-point minimum (vector)

This instruction compares corresponding elements in the vectors in the two source SIMD&FP registers, places the smaller of each of the two floating-point values into a vector, and writes the vector to the destination SIMD&FP register.

When FPCR.AH is 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows:

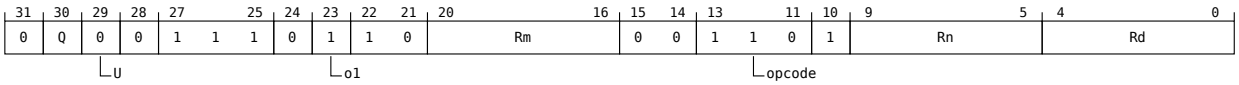
- If both elements are zeros, regardless of the sign of either zero, the result is the second element.
- If either element is a NaN, regardless of the value of FPCR.DN, the result is the second element.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(FEAT_AdvSIMD && FEAT_FP16)



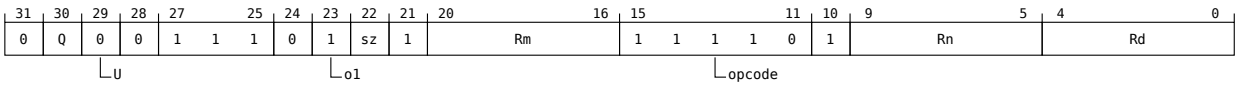
Encoding

```
FMIN <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
FMIN <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the "Half-precision" variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the "Single-precision and double-precision" variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPMin(element1, element2, FPCR);
V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.105 FMIN (scalar)

Floating-point minimum (scalar)

This instruction compares the first and second source SIMD&FP register values, and writes the smaller of the two floating-point values to the destination SIMD&FP register.

When FPCR.AH is 0, the behavior is as follows:

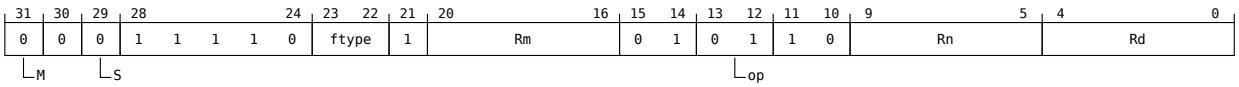
- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either value is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either value is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows:

- If both values are zeros, regardless of the sign of either zero, the result is the second value.
- If either value is a NaN, regardless of the value of FPCR.DN, the result is the second value.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Encoding for the Half-precision variant
(FEAT_FP16) Applies when `(ftype == 11)`

```
FMIN <Hd>, <Hn>, <Hm>
```

Encoding for the Single-precision variant
(FEAT_FP) Applies when `(ftype == 00)`

```
FMIN <Sd>, <Sn>, <Sm>
```

Encoding for the Double-precision variant
(FEAT_FP) Applies when `(ftype == 01)`

```
FMIN <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if ftype == '10' then EndOfDecode(Decode_UNDEF);
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 8 << UInt(ftype EOR '10');
```

Assembler Symbols

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Hm**> Is the 16-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<**Sd**> Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Sn**> Is the 32-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Sm**> Is the 32-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<**Dd**> Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Dn**> Is the 64-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Dm**> Is the 64-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(esize) operand1 = V[n, esize];
constant bits(esize) operand2 = V[m, esize];

V[d, esize] = FPMin(operand1, operand2, FPCR);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.106 FMINNM (vector)

Floating-point minimum number (vector)

This instruction compares corresponding vector elements in the two source SIMD&FP registers, writes the smaller of the two floating-point values into a vector, and writes the vector to the destination SIMD&FP register.

Regardless of the value of FPCR.AH, the behavior is as follows:

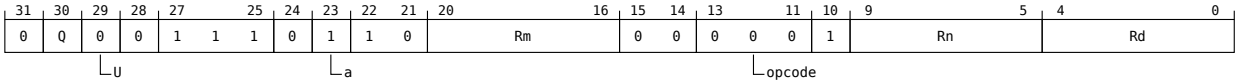
- Negative zero compares less than positive zero.
- If one element is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either element is a signaling NaN or if both elements are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a signaling NaN or if both elements are NaNs, the result is Default NaN.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(FEAT_AdvSIMD && FEAT_FP16)



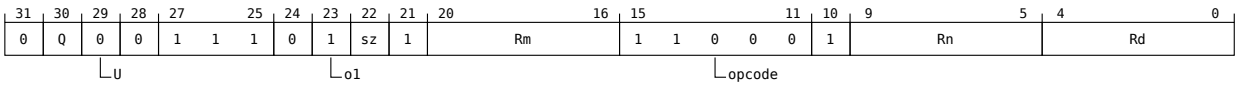
Encoding

```
FMINNM <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

```
FMINNM <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> For the "Half-precision" variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the "Single-precision and double-precision" variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPMinNum(element1, element2, FPCR);
V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.107 FMINNM (scalar)

Floating-point minimum number (scalar)

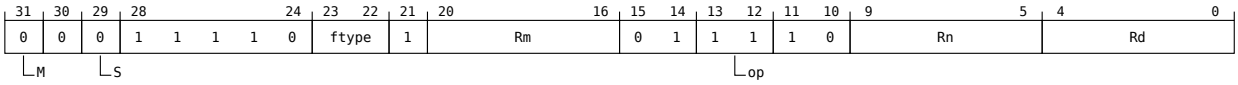
This instruction compares the first and second source SIMD&FP register values, and writes the smaller of the two floating-point values to the destination SIMD&FP register.

Regardless of the value of FPCR.AH, the behavior is as follows:

- Negative zero compares less than positive zero.
- If one value is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either value is a signaling NaN or if both values are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either value is a signaling NaN or if both values are NaNs, the result is Default NaN.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Encoding for the Half-precision variant
(FEAT_FP16) Applies when `(ftype == 11)`

```
FMINNM <Hd>, <Hn>, <Hm>
```

Encoding for the Single-precision variant
(FEAT_FP) Applies when `(ftype == 00)`

```
FMINNM <Sd>, <Sn>, <Sm>
```

Encoding for the Double-precision variant
(FEAT_FP) Applies when `(ftype == 01)`

```
FMINNM <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if ftype == '10' then EndOfDecode(Decode_UNDEF);
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 8 << UInt(ftype EOR '10');
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Hm**> Is the 16-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**Sd**> Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Sn**> Is the 32-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Sm**> Is the 32-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<**Dd**> Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Dn**> Is the 64-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Dm**> Is the 64-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(esize) operand1 = V[n, esize];
constant bits(esize) operand2 = V[m, esize];

V[d, esize] = FPMinNum(operand1, operand2, FPCR);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.108 FMINNMP (scalar)

Floating-point minimum number of pair of elements (scalar)

This instruction compares two vector elements in the source SIMD&FP register and writes the smallest of the floating-point values as a scalar to the destination SIMD&FP register.

Regardless of the value of FPCR.AH, the behavior is as follows for each pairwise operation:

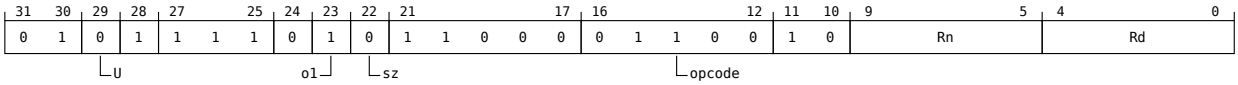
- Negative zero compares less than positive zero.
- If one element is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either element is a signaling NaN or if both elements are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a signaling NaN or if both elements are NaNs, the result is Default NaN.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(FEAT_AdvSIMD && FEAT_FP16)



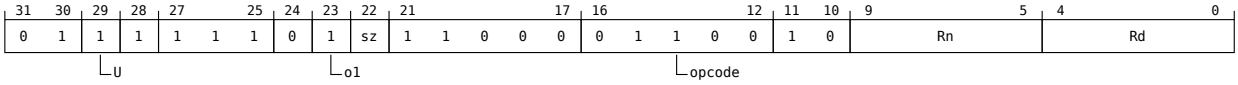
Encoding

```
FMINNMP H<d>, <Vn>.2H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
if sz == '1' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 16;
constant integer datasize = 32;
```

Single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

```
FMINNMP <V><d>, <Vn>.<T>
```

Decode for this encoding

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 32 << UInt(sz);
constant integer datasize = esize * 2;
```

Assembler Symbols

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is the destination width specifier, encoded in "sz":

sz	<V>
0	S
1	D

<T> Is the source arrangement specifier, encoded in "sz":

sz	<T>
0	2S
1	2D

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
constant bits(esize) element1 = Elem[operand, 0, esize];
constant bits(esize) element2 = Elem[operand, 1, esize];
V[d, esize] = FPMinNum(element1, element2, FPCR);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.109 FMINNMP (vector)

Floating-point minimum number pairwise (vector)

This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the smallest of each pair of floating-point values into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

Regardless of the value of FPCR.AH, the behavior is as follows for each pairwise operation:

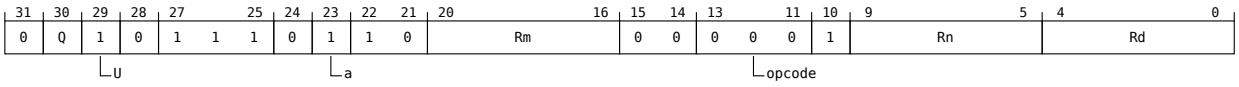
- Negative zero compares less than positive zero.
- If one element is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either element is a signaling NaN or if both elements are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a signaling NaN or if both elements are NaNs, the result is Default NaN.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(FEAT_AdvSIMD && FEAT_FP16)



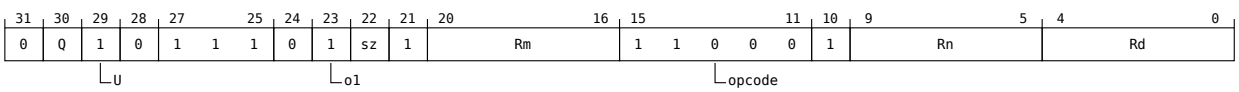
Encoding

```
FMINNMP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
FMINNMP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the "Half-precision" variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the "Single-precision and double-precision" variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
constant bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[concat, 2*e, esize];
    element2 = Elem[concat, (2*e)+1, esize];
    Elem[result, e, esize] = FPMinNum(element1, element2, FPCR);
V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.110 FMINNMV

Floating-point minimum number across vector

This instruction compares all the vector elements in the source SIMD&FP register, and writes the smallest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are floating-point values.

Regardless of the value of FPCR.AH, the behavior is as follows:

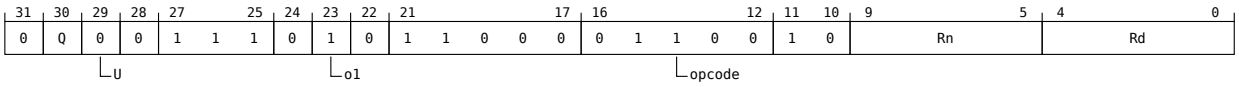
- Negative zero compares less than positive zero.
- If one value is numeric and the other is a quiet NaN, the result is the numeric value.
- When FPCR.DN is 0, if either value is a signaling NaN or if both values are NaNs, the result is a quiet NaN.
- When FPCR.DN is 1, if either value is a signaling NaN or if both values are NaNs, the result is Default NaN.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision

Half-precision
(FEAT_AdvSIMD && FEAT_FP16)



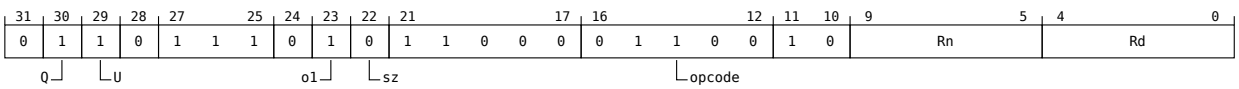
Encoding

```
FMINNMV <V><d>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
```

Single-precision
(FEAT_AdvSIMD)



Encoding

```
FMINNMV S<d>, <Vn>.4S
```

Decode for this encoding

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q != '01' then EndOfDecode(Decode_UNDEF);        // .4S only

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 32;
constant integer datasize = 64 << UInt(Q);
```

Assembler Symbols

<**V**> Is the destination width specifier, H.

<**d**> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<**Vn**> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<**T**> Is an arrangement specifier, encoded in “Q”:

Q	<**T**>
0	4H
1	8H

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
V[d, esize] = FPReduce(ReduceOp_FMINNUM, operand, esize, FPCR);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.111 FMINP (scalar)

Floating-point minimum of pair of elements (scalar)

This instruction compares two vector elements in the source SIMD&FP register and writes the smallest of the floating-point values as a scalar to the destination SIMD&FP register.

When FPCR.AH is 0, the behavior is as follows for each pairwise operation:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows for each pairwise operation:

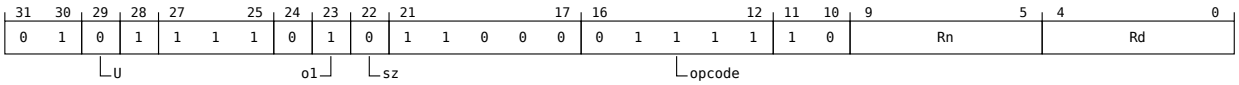
- If both elements are zeros, regardless of the sign of either zero, the result is the second element.
- If either element is a NaN, regardless of the value of FPCR.DN, the result is the second element.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(FEAT_AdvSIMD && FEAT_FP16)



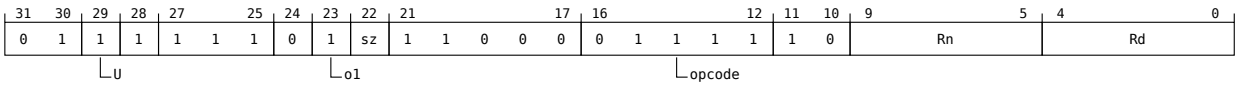
Encoding

```
FMINP H<d>, <Vn>.2H
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
if sz == '1' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 16;
constant integer datasize = 32;
```

Single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
FMINP <V><d>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 32 << UInt(sz);
constant integer datasize = esize * 2;
```

Assembler Symbols

<**d**> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<**Vn**> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<**V**> Is the destination width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<**T**> Is the source arrangement specifier, encoded in “sz”:

sz	<T>
0	2S
1	2D

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
constant bits(esize) element1 = Elem[operand, 0, esize];
constant bits(esize) element2 = Elem[operand, 1, esize];
V[d, esize] = FPMin(element1, element2, FPCR);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.112 FMINP (vector)

Floating-point minimum pairwise (vector)

This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements from the concatenated vector, writes the smaller of each pair of values into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

When FPCR.AH is 0, the behavior is as follows for each pairwise operation:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either element is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either element is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows for each pairwise operation:

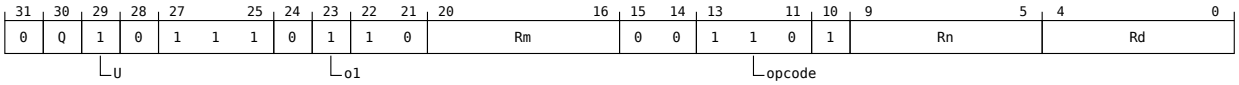
- If both elements are zeros, regardless of the sign of either zero, the result is the second element.
- If either element is a NaN, regardless of the value of FPCR.DN, the result is the second element.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(FEAT_AdvSIMD && FEAT_FP16)



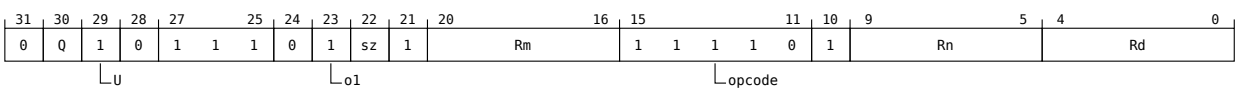
Encoding

```
FMINP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Single-precision and double-precision
(FEAT_AdvSIMD)



Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding

```
FMINP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the "Half-precision" variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the "Single-precision and double-precision" variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
constant bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[concat, 2*e, esize];
    element2 = Elem[concat, (2*e)+1, esize];
    Elem[result, e, esize] = FPMin(element1, element2, FPCR);
V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.113 FMINV

Floating-point minimum across vector

This instruction compares all the vector elements in the source SIMD&FP register, and writes the smallest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are floating-point values.

When FPCR.AH is 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either value is a NaN, the result is a quiet NaN.
- When FPCR.DN is 1, if either value is a NaN, the result is Default NaN.

When FPCR.AH is 1, the behavior is as follows:

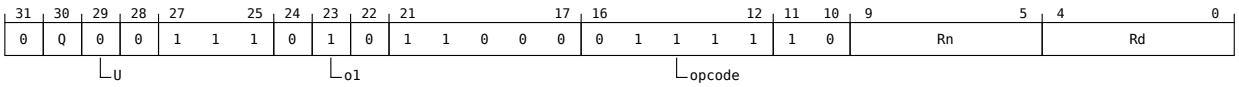
- If both values are zeros, regardless of the sign of either zero, the result is the second value.
- If either value is a NaN, regardless of the value of FPCR.DN, the result is the second value.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision

Half-precision
(FEAT_AdvSIMD && FEAT_FP16)



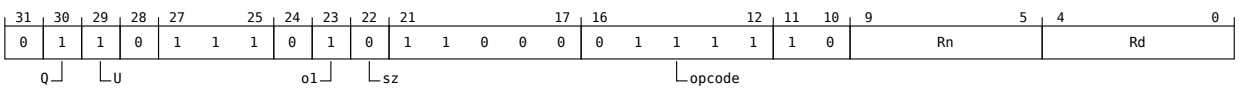
Encoding

```
FMINV <V><d>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
```

Single-precision
(FEAT_AdvSIMD)



Encoding

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
FMINV S<d>, <Vn>.4S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q != '01' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 32;
constant integer datasize = 64 << UInt(Q);
```

Assembler Symbols

<V> Is the destination width specifier, H.

<d> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<T> Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
V[d, esize] = FPReduce(ReduceOp_FMIN, operand, esize, FPCR);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.114 FMLA (by element)

Floating-point fused multiply-add to accumulator (by element)

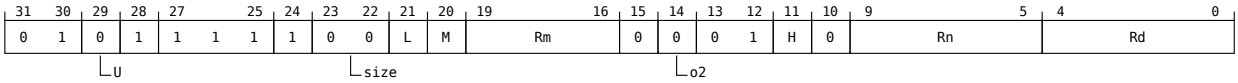
This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the results in the vector elements of the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar, half-precision , Scalar, single-precision and double-precision , Vector, half-precision and Vector, single-precision and double-precision

Scalar, half-precision
(FEAT_AdvSIMD && FEAT_FP16)



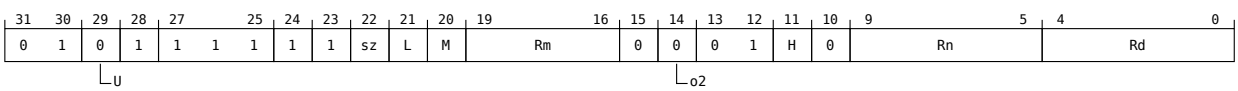
Encoding

```
FMLA <Hd>, <Hn>, <Vm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);

constant integer idxdsize = 64 << UInt(H);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer d = UInt(Rd);
constant integer index = UInt(H:L:M);

constant integer esize = 16;
constant integer datasize = esize;
constant integer elements = 1;
```

Scalar, single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

```
FMLA <V><d>, <V><n>, <Vm>.<Ts>[<index>]
```

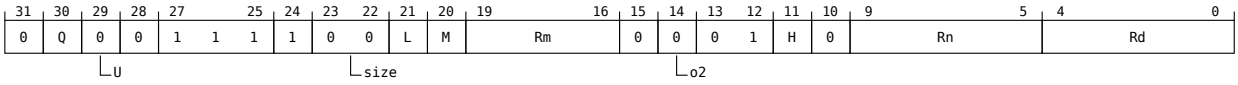
Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer idxdsize = 64 << UInt(H);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
integer index;
constant bit Rmhi = M;
case sz:L of
    when '0x' index = UInt(H:L);
    when '10' index = UInt(H);
    when '11' EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rmhi:Rm);
constant integer esize = 32 << UInt(sz);
constant integer datasize = esize;
constant integer elements = 1;
```

Vector, half-precision (FEAT_AdvSIMD && FEAT_FP16)



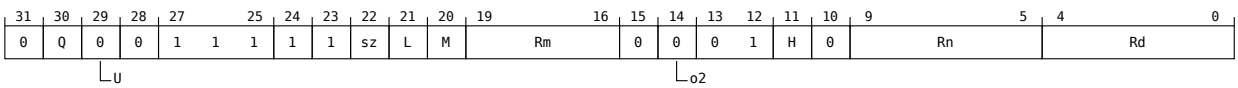
Encoding

```
FMLA <Vd>.<T>, <Vn>.<T>, <Vm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);

constant integer idxdsize = 64 << UInt(H);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer d = UInt(Rd);
constant integer index = UInt(H:L:M);

constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Vector, single-precision and double-precision (FEAT_AdvSIMD)



Encoding

```
FMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer idxdsize = 64 << UInt(H);
integer index;
constant bit Rmhi = M;
case sz:L of
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    when '0x' index = UInt(H:L);
    when '10' index = UInt(H);
    when '11' EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rmhi:Rm);
constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<**Hn**> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<**Vm**> For the "Scalar, half-precision" and "Vector, half-precision" variants: is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the "Rm" field.

For the "Scalar, single-precision and double-precision" and "Vector, single-precision and double-precision" variants: is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

<**index**> For the "Scalar, half-precision" and "Vector, half-precision" variants: is the element index, in the range 0 to 7, encoded in the "H:L:M" fields.

For the "Scalar, single-precision and double-precision" and "Vector, single-precision and double-precision" variants: is the element index, encoded in "sz:L:H":

sz	L	<index>
0	x	UInt(H:L)
1	0	UInt(H)
1	1	RESERVED

<**V**> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**n**> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<**Ts**> Is an element size specifier, encoded in "sz":

sz	<Ts>
0	S
1	D

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> For the "Vector, half-precision" variant: is an arrangement specifier, encoded in "Q":

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q	<T>
0	4H
1	8H

For the “Vector, single-precision and double-precision” variant: is an arrangement specifier, encoded in “Q:sz”:

Q	sz	<T>
0	0	2S
0	1	RESERVED
1	0	4S
1	1	2D

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(idxdsize) operand2 = V[m, idxdsize];
constant bits(datasize) operand3 = V[d, datasize];
bits(esize) element1;
constant bits(esize) element2 = Elem[operand2, index, esize];
bits(datasize) result;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    Elem[result, e, esize] = FPMulAdd(Elem[operand3, e, esize], element1, element2, FPCR);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.115 FMLA (vector)

Floating-point fused multiply-add to accumulator (vector)

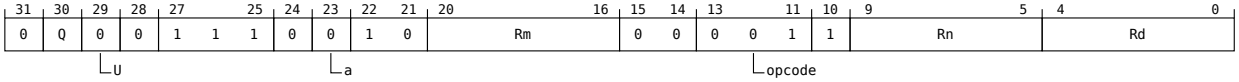
This instruction multiplies corresponding floating-point values in the vectors in the two source SIMD&FP registers, adds the product to the corresponding vector element of the destination SIMD&FP register, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(FEAT_AdvSIMD && FEAT_FP16)



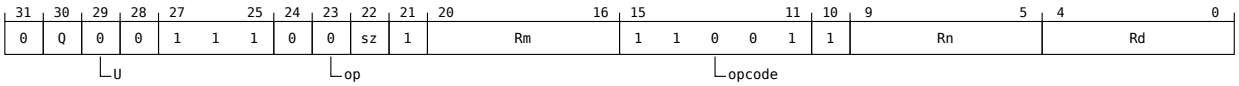
Encoding

```
FMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

```
FMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> For the "Half-precision" variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the "Single-precision and double-precision" variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
constant bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPMulAdd(Elem[operand3, e, esize], element1, element2, FPCR);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.116 FMLAL, FMLAL2 (by element)

Floating-point fused multiply-add long to accumulator (by element)

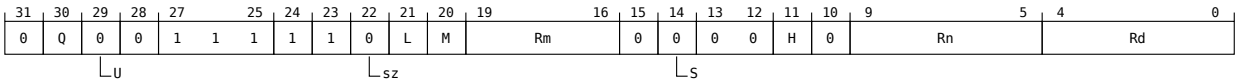
This instruction multiplies the half-precision vector elements in the first source SIMD&FP register by the specified half-precision value in the second source SIMD&FP register, and accumulates the intermediate product without rounding to the corresponding single-precision vector element of the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: FMLAL and FMLAL2

FMLAL
(FEAT_FHM)



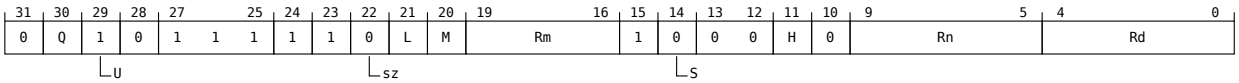
Encoding

```
FMLAL <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FHM) then EndOfDecode(Decode_UNDEF);
if sz == '1' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt('0':Rm);    // Vm can only be in bottom 16 registers.
constant integer index = UInt(H:L:M);

constant integer esize = 32;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;

constant integer part = 0;
```

FMLAL2
(FEAT_FHM)



Encoding

```
FMLAL2 <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FHM) then EndOfDecode(Decode_UNDEF);
if sz == '1' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer n = UInt(Rn);
constant integer m = UInt('0':Rm);    // Vm can only be in bottom 16 registers.
constant integer index = UInt(H:L:M);

constant integer esize = 32;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;

constant integer part = 1;
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**Ta**> Is an arrangement specifier, encoded in "Q":

Q	<Ta>
0	2S
1	4S

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<**Tb**> Is an arrangement specifier, encoded in "Q":

Q	<Tb>
0	2H
1	4H

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<**index**> Is the element index, encoded in the "H:L:M" fields.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize DIV 2) operand1 = Vpart[n, part, datasize DIV 2];
constant bits(128) operand2 = V[m, 128];
constant bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
bits(esize DIV 2) element1;
constant bits(esize DIV 2) element2 = Elem[operand2, index, esize DIV 2];

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize DIV 2];
    Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2, FPCR);
V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.117 FMLAL, FMLAL2 (vector)

Floating-point fused multiply-add long to accumulator (vector)

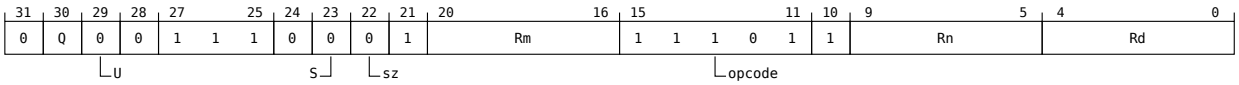
This instruction multiplies corresponding half-precision floating-point values in the vectors in the two source SIMD&FP registers, and accumulates the intermediate product without rounding to the corresponding single-precision vector element of the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: FMLAL and FMLAL2

FMLAL
(FEAT_FHM)



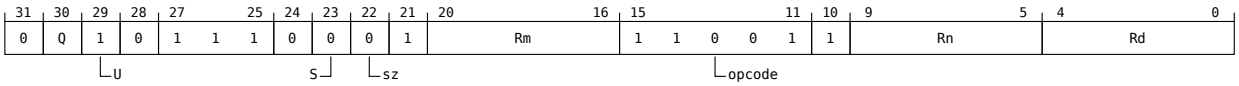
Encoding

```
FMLAL <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FHM) then EndOfDecode(Decode_UNDEF);
if sz == '1' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 32;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
constant integer part = 0;
```

FMLAL2
(FEAT_FHM)



Encoding

```
FMLAL2 <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FHM) then EndOfDecode(Decode_UNDEF);
if sz == '1' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 32;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
constant integer part = 1;
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**Ta**> Is an arrangement specifier, encoded in "Q":

Q	<Ta>
0	2S
1	4S

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<**Tb**> Is an arrangement specifier, encoded in "Q":

Q	<Tb>
0	2H
1	4H

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize DIV 2) operand1 = Vpart[n, part, datasize DIV 2];
constant bits(datasize DIV 2) operand2 = Vpart[m, part, datasize DIV 2];
constant bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize DIV 2];
    element2 = Elem[operand2, e, esize DIV 2];
    Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2, FPCR);
V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.118 FMLS (by element)

Floating-point fused multiply-subtract from accumulator (by element)

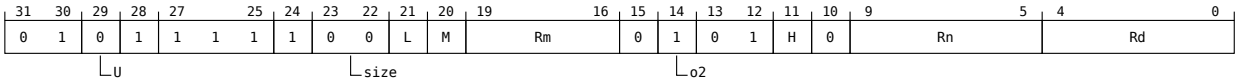
This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and subtracts the results from the vector elements of the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar, half-precision , Scalar, single-precision and double-precision , Vector, half-precision and Vector, single-precision and double-precision

Scalar, half-precision
(FEAT_AdvSIMD && FEAT_FP16)



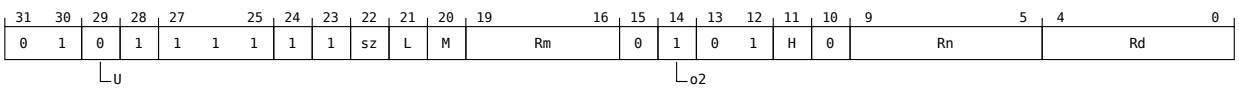
Encoding

```
FMLS <Hd>, <Hn>, <Vm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);

constant integer idxdsize = 64 << UInt(H);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer d = UInt(Rd);
constant integer index = UInt(H:L:M);

constant integer esize = 16;
constant integer datasize = esize;
constant integer elements = 1;
```

Scalar, single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

```
FMLS <V><d>, <V><n>, <Vm>.<Ts>[<index>]
```

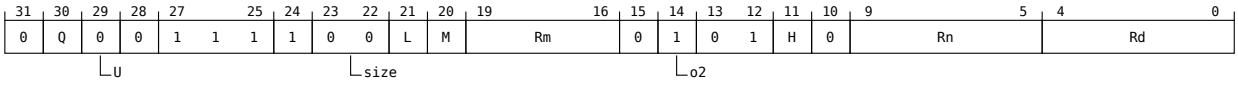
Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer idxdsize = 64 << UInt(H);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
integer index;
constant bit Rmhi = M;
case sz:L of
    when '0x' index = UInt(H:L);
    when '10' index = UInt(H);
    when '11' EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rmhi:Rm);
constant integer esize = 32 << UInt(sz);
constant integer datasize = esize;
constant integer elements = 1;
```

Vector, half-precision (FEAT_AdvSIMD && FEAT_FP16)



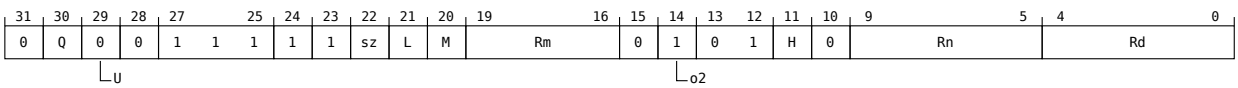
Encoding

```
FMLS <Vd>.<T>, <Vn>.<T>, <Vm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);

constant integer idxdsize = 64 << UInt(H);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer d = UInt(Rd);
constant integer index = UInt(H:L:M);

constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Vector, single-precision and double-precision (FEAT_AdvSIMD)



Encoding

```
FMLS <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer idxdsize = 64 << UInt(H);
integer index;
constant bit Rmhi = M;
case sz:L of
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    when '0x' index = UInt(H:L);
    when '10' index = UInt(H);
    when '11' EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rmhi:Rm);
constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<**Hn**> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<**Vm**> For the "Scalar, half-precision" and "Vector, half-precision" variants: is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the "Rm" field.

For the "Scalar, single-precision and double-precision" and "Vector, single-precision and double-precision" variants: is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

<**index**> For the "Scalar, half-precision" and "Vector, half-precision" variants: is the element index, in the range 0 to 7, encoded in the "H:L:M" fields.

For the "Scalar, single-precision and double-precision" and "Vector, single-precision and double-precision" variants: is the element index, encoded in "sz:L:H":

sz	L	<index>
0	x	UInt(H:L)
1	0	UInt(H)
1	1	RESERVED

<**V**> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**n**> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<**Ts**> Is an element size specifier, encoded in "sz":

sz	<Ts>
0	S
1	D

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> For the "Vector, half-precision" variant: is an arrangement specifier, encoded in "Q":

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Q	<T>
0	4H
1	8H

For the “Vector, single-precision and double-precision” variant: is an arrangement specifier, encoded in “Q:sz”:

Q	sz	<T>
0	0	2S
0	1	RESERVED
1	0	4S
1	1	2D

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(idxdsize) operand2 = V[m, idxdsize];
constant bits(datasize) operand3 = V[d, datasize];
bits(esize) element1;
constant bits(esize) element2 = Elem[operand2, index, esize];
bits(datasize) result;

for e = 0 to elements-1
    element1 = FPNeg(Elem[operand1, e, esize], FPCR);
    Elem[result, e, esize] = FPMulAdd(Elem[operand3, e, esize], element1, element2, FPCR);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.119 FMLS (vector)

Floating-point fused multiply-subtract from accumulator (vector)

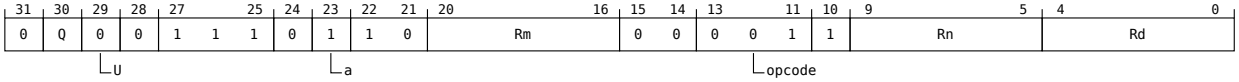
This instruction multiplies corresponding floating-point values in the vectors in the two source SIMD&FP registers, negates the product, adds the result to the corresponding vector element of the destination SIMD&FP register, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(FEAT_AdvSIMD && FEAT_FP16)



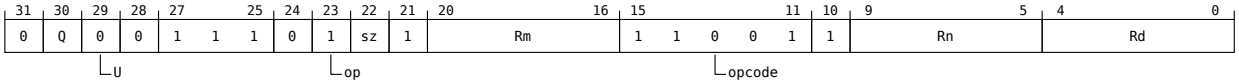
Encoding

```
FMLS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

```
FMLS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> For the "Half-precision" variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the "Single-precision and double-precision" variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
constant bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = FPNeg(Elem[operand1, e, esize], FPCR);
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPMulAdd(Elem[operand3, e, esize], element1, element2, FPCR);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.120 FMLSL, FMLSL2 (by element)

Floating-point fused multiply-subtract long from accumulator (by element)

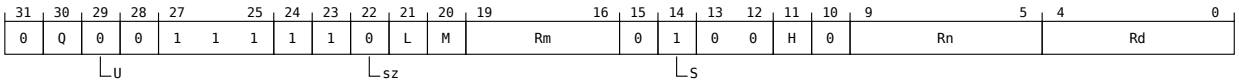
This instruction multiplies the negated half-precision vector elements in the first source SIMD&FP register by the specified half-precision value in the second source SIMD&FP register, and accumulates the intermediate product without rounding to the corresponding single-precision vector element of the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: FMLSL and FMLSL2

FMLSL
(FEAT_FHM)



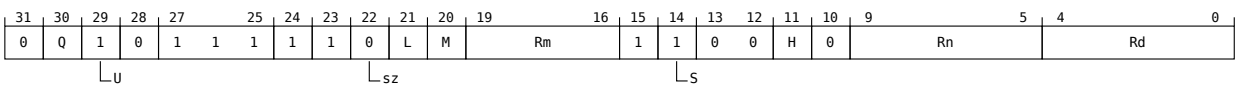
Encoding

```
FMLSL <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FHM) then EndOfDecode(Decode_UNDEF);
if sz == '1' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt('0':Rm);    // Vm can only be in bottom 16 registers.
constant integer index = UInt(H:L:M);

constant integer esize = 32;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;

constant integer part = 0;
```

FMLSL2
(FEAT_FHM)



Encoding

```
FMLSL2 <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FHM) then EndOfDecode(Decode_UNDEF);
if sz == '1' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer n = UInt(Rn);
constant integer m = UInt('0':Rm);    // Vm can only be in bottom 16 registers.
constant integer index = UInt(H:L:M);

constant integer esize = 32;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;

constant integer part = 1;
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Ta**> Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	2S
1	4S

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Tb**> Is an arrangement specifier, encoded in “Q”:

Q	<Tb>
0	2H
1	4H

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

<**index**> Is the element index, encoded in the “H:L:M” fields.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize DIV 2) operand1 = Vpart[n, part, datasize DIV 2];
constant bits(128) operand2 = V[m, 128];
constant bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
bits(esize DIV 2) element1;
constant bits(esize DIV 2) element2 = Elem[operand2, index, esize DIV 2];

for e = 0 to elements-1
    element1 = FPNeg(Elem[operand1, e, esize DIV 2], FPCR);
    Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2, FPCR);
V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.121 FMLSL, FMLSL2 (vector)

Floating-point fused multiply-subtract long from accumulator (vector)

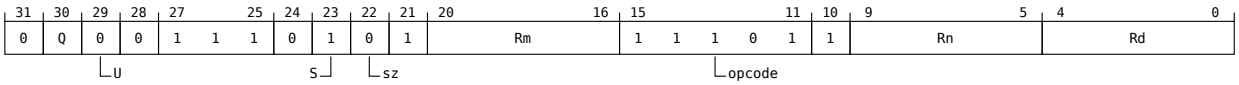
This instruction negates the half-precision values in the vector of one SIMD&FP register, multiplies these with the corresponding half-precision values in another vector, and accumulates the intermediate product without rounding to the corresponding single-precision vector element of the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: FMLSL and FMLSL2

FMLSL
(FEAT_FHM)



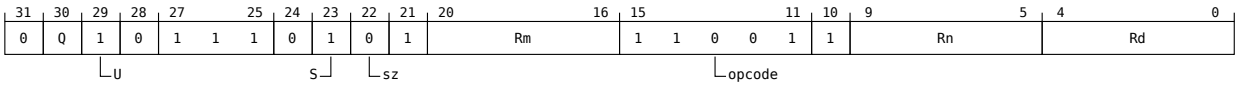
Encoding

```
FMLSL <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FHM) then EndOfDecode(Decode_UNDEF);
if sz == '1' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 32;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
constant integer part = 0;
```

FMLSL2
(FEAT_FHM)



Encoding

```
FMLSL2 <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FHM) then EndOfDecode(Decode_UNDEF);
if sz == '1' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 32;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
constant integer part = 1;
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**Ta**> Is an arrangement specifier, encoded in "Q":

Q	<Ta>
0	2S
1	4S

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<**Tb**> Is an arrangement specifier, encoded in "Q":

Q	<Tb>
0	2H
1	4H

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize DIV 2) operand1 = Vpart[n, part, datasize DIV 2];
constant bits(datasize DIV 2) operand2 = Vpart[m, part, datasize DIV 2];
constant bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2;

for e = 0 to elements-1
    element1 = FPNeg(Elem[operand1, e, esize DIV 2], FPCR);
    element2 = Elem[operand2, e, esize DIV 2];
    Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2, FPCR);
V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.122 FMOV (vector, immediate)

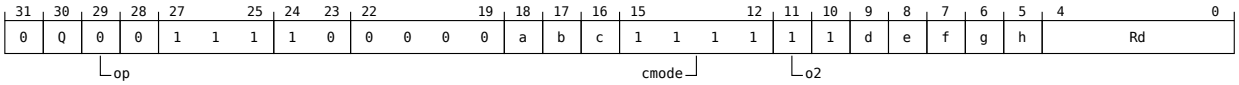
Floating-point move immediate (vector)

This instruction copies an immediate floating-point constant into every element of the SIMD&FP destination register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(FEAT_AdvSIMD && FEAT_FP16)



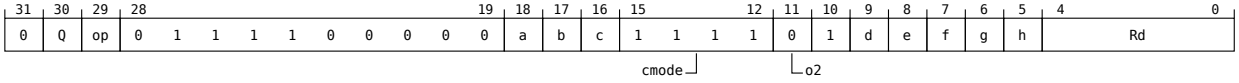
Encoding

```
FMOV <Vd>.<T>, #<imm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);

constant integer rd = UInt(Rd);

constant integer datasize = 64 << UInt(Q);
constant bits(8) imm8 = a:b:c:d:e:f:g:h;
constant bits(16) imm16 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>, 2):imm8<5:0>:Zeros(6);
constant bits(datasize) imm = Replicate(imm16, datasize DIV 16);
```

Single-precision and double-precision
(FEAT_AdvSIMD)



Encoding for the Single-precision variant Applies when `(op == 0)`

```
FMOV <Vd>.<T>, #<imm>
```

Encoding for the Double-precision variant Applies when `(Q == 1 && op == 1)`

```
FMOV <Vd>.2D, #<imm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if cmode:op == '11111' then
    // FMOV Dn,#imm is in main FP instruction set
    if Q == '0' then EndOfDecode(Decode_UNDEF);

constant integer rd = UInt(Rd);
constant integer datasize = 64 << UInt(Q);
constant bits(64) imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
constant bits(datasize) imm = Replicate(imm64, datasize DIV 64);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> For the "Half-precision" variant: is an arrangement specifier, encoded in "Q":

Q	<**T**>
0	4H
1	8H

For the "Single-precision" variant: is an arrangement specifier, encoded in "Q":

Q	<**T**>
0	2S
1	4S

<**imm**> Is a signed floating-point constant with 3-bit exponent and normalized 4 bits of precision, encoded in "a:b:c:d:e:f:g:h". For details of the range of constants available and the encoding of <imm>, see Modified immediate constants in A64 floating-point instructions.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
V[rd, datasize] = imm;
```

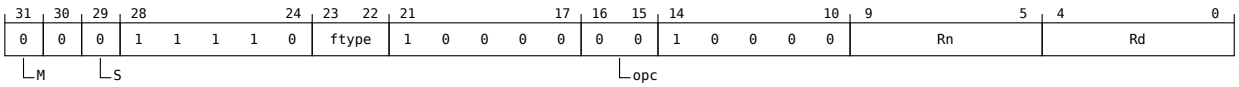

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.123 FMOV (register)

Floating-point move register without conversion

This instruction copies the floating-point value in the SIMD&FP source register to the SIMD&FP destination register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Encoding for the Half-precision variant
(FEAT_FP16) Applies when `(ftype == 11)`

```
FMOV <Hd>, <Hn>
```

Encoding for the Single-precision variant
(FEAT_FP) Applies when `(ftype == 00)`

```
FMOV <Sd>, <Sn>
```

Encoding for the Double-precision variant
(FEAT_FP) Applies when `(ftype == 01)`

```
FMOV <Dd>, <Dn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if ftype == '10' then EndOfDecode(Decode_UNDEF);
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 8 << UInt(ftype EOR '10');
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(esize) operand = V[n, esize];

V[d, esize] = operand;
```

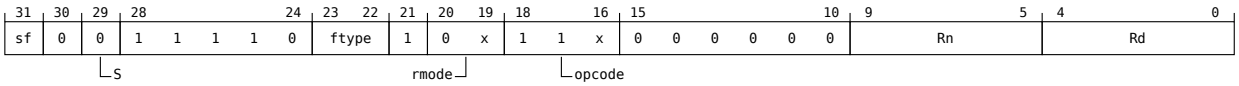

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.124 FMOV (general)

Floating-point move to or from general-purpose register without conversion

This instruction transfers the contents of a SIMD&FP register to a general-purpose register, or the contents of a general-purpose register to a SIMD&FP register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Encoding for the Half-precision to 32-bit variant
(FEAT_FP16) Applies when `(sf == 0 && ftype == 11 && rmode == 00 && opcode == 110)`

```
FMOV <Wd>, <Hn>
```

Encoding for the Half-precision to 64-bit variant
(FEAT_FP16) Applies when `(sf == 1 && ftype == 11 && rmode == 00 && opcode == 110)`

```
FMOV <Xd>, <Hn>
```

Encoding for the 32-bit to half-precision variant
(FEAT_FP16) Applies when `(sf == 0 && ftype == 11 && rmode == 00 && opcode == 111)`

```
FMOV <Hd>, <Wn>
```

Encoding for the 32-bit to single-precision variant
(FEAT_FP) Applies when `(sf == 0 && ftype == 00 && rmode == 00 && opcode == 111)`

```
FMOV <Sd>, <Wn>
```

Encoding for the Single-precision to 32-bit variant
(FEAT_FP) Applies when `(sf == 0 && ftype == 00 && rmode == 00 && opcode == 110)`

```
FMOV <Wd>, <Sn>
```

Encoding for the 64-bit to half-precision variant
(FEAT_FP16) Applies when `(sf == 1 && ftype == 11 && rmode == 00 && opcode == 111)`

```
FMOV <Hd>, <Xn>
```

Encoding for the 64-bit to double-precision variant
(FEAT_FP) Applies when `(sf == 1 && ftype == 01 && rmode == 00 && opcode == 111)`

```
FMOV <Dd>, <Xn>
```

Encoding for the 64-bit to top half of 128-bit variant
(FEAT_FP) Applies when `(sf == 1 && ftype == 10 && rmode == 01 && opcode == 111)`

```
FMOV <Vd>.D[1], <Xn>
```

Encoding for the Double-precision to 64-bit variant
(FEAT_FP) Applies when `(sf == 1 && ftype == 01 && rmode == 00 && opcode == 110)`

```
FMOV <Xd>, <Dn>
```

Encoding for the Top half of 128-bit to 64-bit variant
(FEAT_FP) Applies when `(sf == 1 && ftype == 10 && rmode == 01 && opcode == 110)`

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
FMOV <Xd>, <Vn>.D[1]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if ftype == '10' && opcode<2:1>:rmode != '11 01' then EndOfDecode(Decode_UNDEF);
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer intsize = 32 << UInt(sf);
constant integer fltsize = if ftype == '10' then 64 else (8 << UInt(ftype EOR '10'));
constant integer part = UInt(rmode<0>);
FPConvOp op;

case opcode<2:1>:rmode of
    when '11 00'        // FMOV
        if fltsize != 16 && fltsize != intsize then EndOfDecode(Decode_UNDEF);
        op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    when '11 01'        // FMOV D[1]
        if intsize != 64 || ftype != '10' then EndOfDecode(Decode_UNDEF);
        op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
    otherwise
        Unreachable();
```

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Wn> Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Xn> Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<Vn> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(fltsize) fltval;
bits(intsize) intval;

case op of
    when FPConvOp_MOV_FtoI
        fltval = Vpart[n, part, fltsize];
        X[d, intsize] = ZeroExtend(fltval, intsize);
    when FPConvOp_MOV_ItoF
        intval = X[n, intsize];
        Vpart[d, part, fltsize] = intval<fltsize-1:0>;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    otherwise
        Unreachable();
```

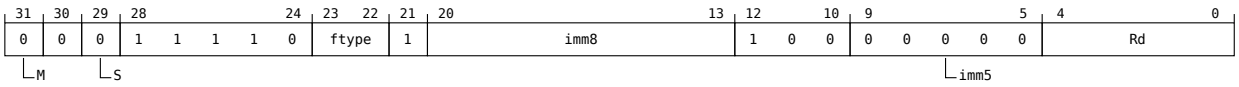

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.125 FMOV (scalar, immediate)

Floating-point move immediate (scalar)

This instruction copies a floating-point immediate constant into the SIMD&FP destination register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Encoding for the Half-precision variant
(FEAT_FP16) Applies when `(ftype == 11)`

```
FMOV <Hd>, #<imm>
```

Encoding for the Single-precision variant
(FEAT_FP) Applies when `(ftype == 00)`

```
FMOV <Sd>, #<imm>
```

Encoding for the Double-precision variant
(FEAT_FP) Applies when `(ftype == 01)`

```
FMOV <Dd>, #<imm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if ftype == '10' then EndOfDecode(Decode_UNDEF);
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);

constant integer datasize = 8 << UInt(ftype EOR '10');
constant bits(datasize) imm = VFPExpandImm(imm8, datasize);
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<**imm**> Is a signed floating-point constant with 3-bit exponent and normalized 4 bits of precision, encoded in the "imm8" field. For details of the range of constants available and the encoding of <imm>, see Modified immediate constants in A64 floating-point instructions.

<**Sd**> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<**Dd**> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

V[d, datasize] = imm;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

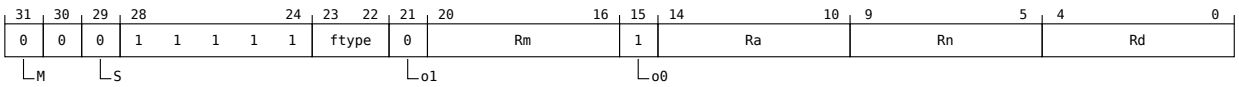
C7.2.126 FMSUB

Floating-point fused multiply-subtract (scalar)

This instruction multiplies the values of the first two SIMD&FP source registers, negates the product, adds that to the value of the third SIMD&FP source register, and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Encoding for the Half-precision variant
(FEAT_FP16) Applies when `(ftype == 11)`

```
FMSUB <Hd>, <Hn>, <Hm>, <Ha>
```

Encoding for the Single-precision variant
(FEAT_FP) Applies when `(ftype == 00)`

```
FMSUB <Sd>, <Sn>, <Sm>, <Sa>
```

Encoding for the Double-precision variant
(FEAT_FP) Applies when `(ftype == 01)`

```
FMSUB <Dd>, <Dn>, <Dm>, <Da>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if ftype == '10' then EndOfDecode(Decode_UNDEF);
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer a = UInt(Ra);

constant integer esize = 8 << UInt(ftype EOR '10');
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the “Rn” field.

<**Hm**> Is the 16-bit name of the second SIMD&FP source register holding the multiplier, encoded in the “Rm” field.

<**Ha**> Is the 16-bit name of the third SIMD&FP source register holding the minuend, encoded in the “Ra” field.

<**Sd**> Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Sn**> Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the “Rn” field.

<**Sm**> Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the “Rm” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**Sa**> Is the 32-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.

<**Dd**> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<**Dn**> Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.

<**Dm**> Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

<**Da**> Is the 64-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(esize) addend   = V[a, esize];
constant bits(esize) operand1 = FPNeg(V[n, esize], FPCR);
constant bits(esize) operand2 = V[m, esize];

V[d, esize] = FPMulAdd(addend, operand1, operand2, FPCR);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.127 FMUL (by element)

Floating-point multiply (by element)

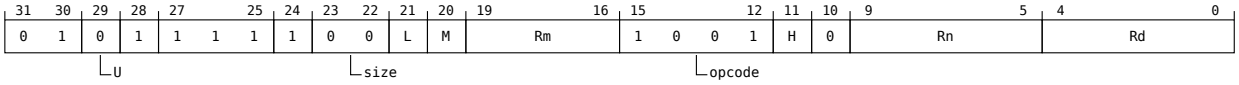
This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar, half-precision , Scalar, single-precision and double-precision , Vector, half-precision and Vector, single-precision and double-precision

Scalar, half-precision
(FEAT_AdvSIMD && FEAT_FP16)



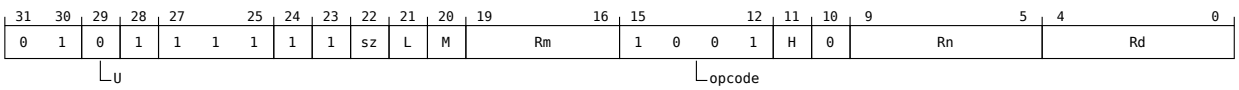
Encoding

```
FMUL <Hd>, <Hn>, <Vm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);

constant integer idxdsize = 64 << UInt(H);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer index = UInt(H:L:M);

constant integer esize = 16;
constant integer datasize = esize;

constant integer elements = 1;
```

Scalar, single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

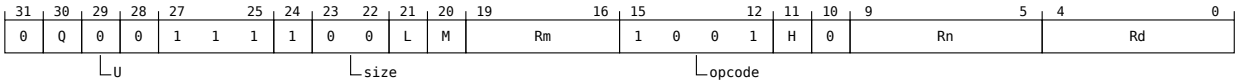
```
FMUL <V><d>, <V><n>, <Vm>.<Ts>[<index>]
```

Decode for this encoding

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer idxdsize = 64 << UInt(H);
integer index;
constant bit Rmhi = M;
case sz:L of
    when '0x' index = UInt(H:L);
    when '10' index = UInt(H);
    when '11' EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rmhi:Rm);

constant integer esize = 32 << UInt(sz);
constant integer datasize = esize;
constant integer elements = 1;
```

Vector, half-precision (FEAT_AdvSIMD && FEAT_FP16)



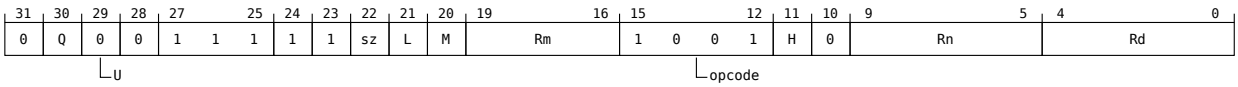
Encoding

```
FMUL <Vd>.<T>, <Vn>.<T>, <Vm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);

constant integer idxdsize = 64 << UInt(H);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer index = UInt(H:L:M);

constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Vector, single-precision and double-precision (FEAT_AdvSIMD)



Encoding

```
FMUL <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer idxdsize = 64 << UInt(H);
integer index;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant bit Rmhi = M;
case sz:L of
    when '0x' index = UInt(H:L);
    when '10' index = UInt(H);
    when '11' EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rmhi:Rm);

if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<**Hn**> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<**Vm**> For the "Scalar, half-precision" and "Vector, half-precision" variants: is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the "Rm" field.

For the "Scalar, single-precision and double-precision" and "Vector, single-precision and double-precision" variants: is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

<**index**> For the "Scalar, half-precision" and "Vector, half-precision" variants: is the element index, in the range 0 to 7, encoded in the "H:L:M" fields.

For the "Scalar, single-precision and double-precision" and "Vector, single-precision and double-precision" variants: is the element index, encoded in "sz:L:H":

sz	L	<index>
0	x	UInt(H:L)
1	0	UInt(H)
1	1	RESERVED

<**V**> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**n**> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<**Ts**> Is an element size specifier, encoded in "sz":

sz	<Ts>
0	S
1	D

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**T**> For the “Vector, half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Vector, single-precision and double-precision” variant: is an arrangement specifier, encoded in “Q:sz”:

Q	sz	<T>
0	0	2S
0	1	RESERVED
1	0	4S
1	1	2D

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(idxdsize) operand2 = V[m, idxdsize];
bits(esize) element1;
constant bits(esize) element2 = Elem[operand2, index, esize];

bits(datasize) result;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    Elem[result, e, esize] = FPMul(element1, element2, FPCR);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.128 FMUL (vector)

Floating-point multiply (vector)

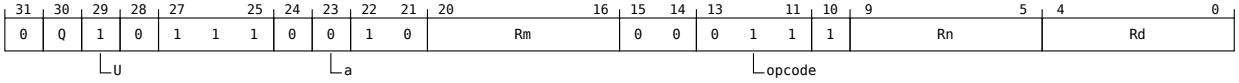
This instruction multiplies corresponding floating-point values in the vectors in the two source SIMD&FP registers, places the result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(FEAT_AdvSIMD && FEAT_FP16)



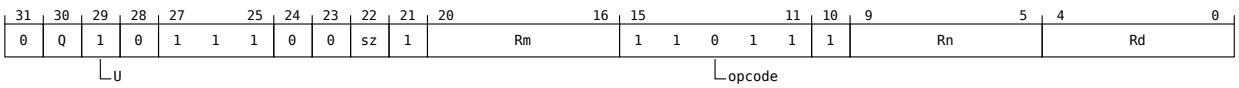
Encoding

```
FMUL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

```
FMUL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> For the "Half-precision" variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the "Single-precision and double-precision" variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPMul(element1, element2, FPCR);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

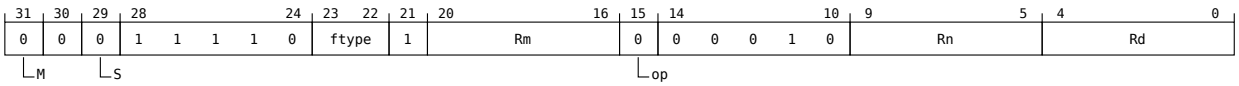
C7.2.129 FMUL (scalar)

Floating-point multiply (scalar)

This instruction multiplies the floating-point values of the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Encoding for the Half-precision variant
(FEAT_FP16) Applies when `(ftype == 11)`

```
FMUL <Hd>, <Hn>, <Hm>
```

Encoding for the Single-precision variant
(FEAT_FP) Applies when `(ftype == 00)`

```
FMUL <Sd>, <Sn>, <Sm>
```

Encoding for the Double-precision variant
(FEAT_FP) Applies when `(ftype == 01)`

```
FMUL <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if ftype == '10' then EndOfDecode(Decode_UNDEF);
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 8 << UInt(ftype EOR '10');
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Hm**> Is the 16-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<**Sd**> Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Sn**> Is the 32-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Sm**> Is the 32-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<**Dd**> Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Dn**> Is the 64-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**Dm**> Is the 64-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(esize) operand1 = V[n, esize];
constant bits(esize) operand2 = V[m, esize];

constant bits(esize) result = FPMul(operand1, operand2, FPCR);
V[d, esize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.130 FMULX (by element)

Floating-point multiply extended (by element)

This instruction multiplies the floating-point values in the vector elements in the first source SIMD&FP register by the specified floating-point value in the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

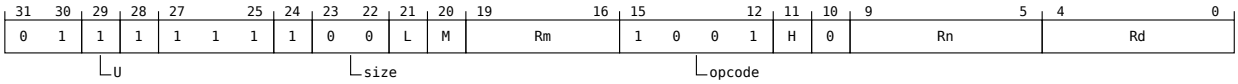
If one value is zero and the other value is infinite, the result is 2.0. In this case, the result is negative if only one of the values is negative, otherwise the result is positive.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar, half-precision , Scalar, single-precision and double-precision , Vector, half-precision and Vector, single-precision and double-precision

Scalar, half-precision
(FEAT_AdvSIMD && FEAT_FP16)



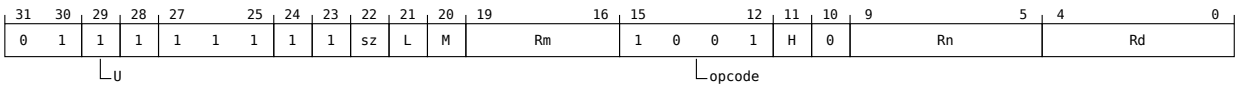
Encoding

```
FMULX <Hd>, <Hn>, <Vm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);

constant integer idxdsize = 64 << UInt(H);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer index = UInt(H:L:M);

constant integer esize = 16;
constant integer datasize = esize;

constant integer elements = 1;
```

Scalar, single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

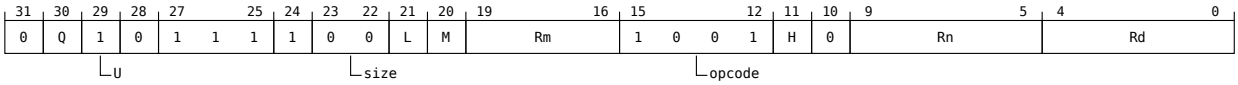
```
FMULX <V><d>, <V><n>, <Vm>.<Ts>[<index>]
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer idxdsize = 64 << UInt(H);
integer index;
constant bit Rmhi = M;
case sz:L of
    when '0x' index = UInt(H:L);
    when '10' index = UInt(H);
    when '11' EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rmhi:Rm);

constant integer esize = 32 << UInt(sz);
constant integer datasize = esize;
constant integer elements = 1;
```

Vector, half-precision (FEAT_AdvSIMD && FEAT_FP16)



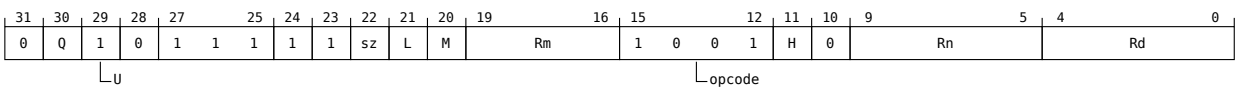
Encoding

```
FMULX <Vd>.<T>, <Vn>.<T>, <Vm>.H[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);

constant integer idxdsize = 64 << UInt(H);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer index = UInt(H:L:M);

constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Vector, single-precision and double-precision (FEAT_AdvSIMD)



Encoding

```
FMULX <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer idxdsize = 64 << UInt(H);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
integer index;
constant bit Rmhi = M;
case sz:L of
    when '0x' index = UInt(H:L);
    when '10' index = UInt(H);
    when '11' EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rmhi:Rm);

if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<**Hn**> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<**Vm**> For the "Scalar, half-precision" and "Vector, half-precision" variants: is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the "Rm" field.

For the "Scalar, single-precision and double-precision" and "Vector, single-precision and double-precision" variants: is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

<**index**> For the "Scalar, half-precision" and "Vector, half-precision" variants: is the element index, in the range 0 to 7, encoded in the "H:L:M" fields.

For the "Scalar, single-precision and double-precision" and "Vector, single-precision and double-precision" variants: is the element index, encoded in "sz:L:H":

sz	L	<index>
0	x	UInt(H:L)
1	0	UInt(H)
1	1	RESERVED

<**V**> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**n**> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<**Ts**> Is an element size specifier, encoded in "sz":

sz	<Ts>
0	S
1	D

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**T**> For the “Vector, half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Vector, single-precision and double-precision” variant: is an arrangement specifier, encoded in “Q:sz”:

Q	sz	<T>
0	0	2S
0	1	RESERVED
1	0	4S
1	1	2D

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(idxdsize) operand2 = V[m, idxdsize];
bits(esize) element1;
constant bits(esize) element2 = Elem[operand2, index, esize];

bits(datasize) result;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    Elem[result, e, esize] = FPMulX(element1, element2, FPCR);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.131 FMULX

Floating-point multiply extended

This instruction multiplies corresponding floating-point values in the vectors of the two source SIMD&FP registers, places the resulting floating-point values in a vector, and writes the vector to the destination SIMD&FP register.

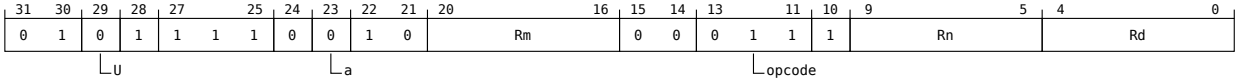
If one value is zero and the other value is infinite, the result is 2.0. In this case, the result is negative if only one of the values is negative, otherwise the result is positive.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision , Scalar single-precision and double-precision , Vector half-precision and Vector single-precision and double-precision

Scalar half-precision
(FEAT_AdvSIMD && FEAT_FP16)



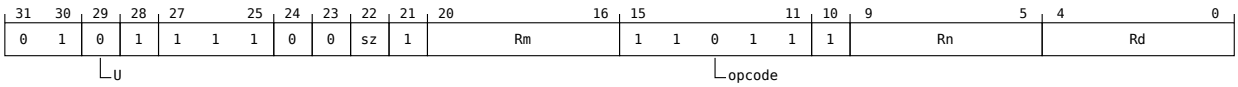
Encoding

```
FMULX <Hd>, <Hn>, <Hm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 16;
constant integer datasize = esize;
constant integer elements = 1;
```

Scalar single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

```
FMULX <V><d>, <V><n>, <V><m>
```

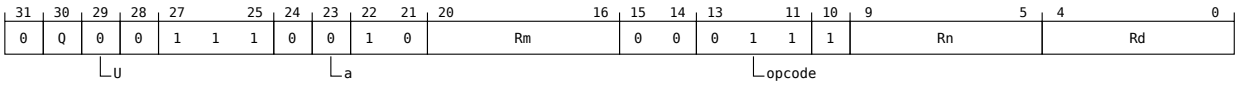
Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer esize = 32 << UInt(sz);
constant integer datasize = esize;
constant integer elements = 1;
```

Vector half-precision (FEAT_AdvSIMD && FEAT_FP16)



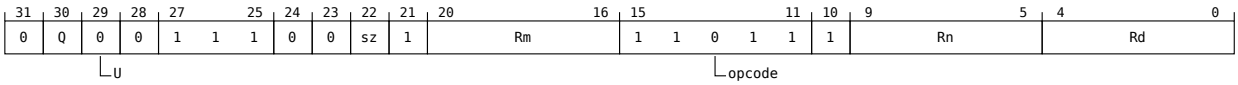
Encoding

```
FMULX <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Vector single-precision and double-precision (FEAT_AdvSIMD)



Encoding

```
FMULX <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Hm**> Is the 16-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<**V**> Is a width specifier, encoded in “sz”:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	<V>
0	S
1	D

<**d**> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<**n**> Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<**m**> Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> For the “Vector half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Vector single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];

bits(esize) element1;
bits(esize) element2;
bits(datasize) result;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPMulX(element1, element2, FPCR);
V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.132 FNEG (vector)

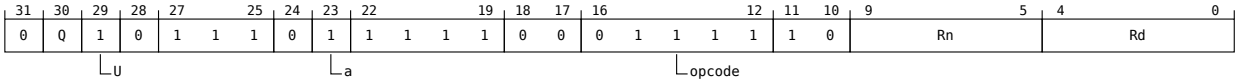
Floating-point negate (vector)

This instruction negates the value of each vector element in the source SIMD&FP register, writes the result to a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(FEAT_AdvSIMD && FEAT_FP16)



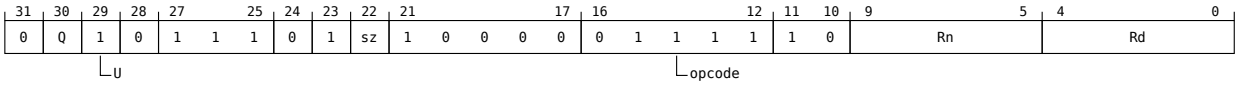
Encoding

```
FNEG <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

```
FNEG <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**T**> For the “Half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(datasize) result;

for e = 0 to elements-1
    constant bits(esize) element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPNeg(element, FPCR);

V[d, datasize] = result;
```

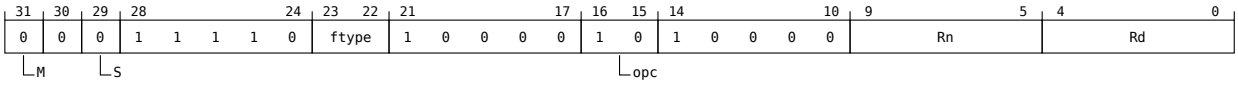

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.133 FNEG (scalar)

Floating-point negate (scalar)

This instruction negates the value in the SIMD&FP source register and writes the result to the SIMD&FP destination register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Encoding for the Half-precision variant
(FEAT_FP16) Applies when `(ftype == 11)`

```
FNEG <Hd>, <Hn>
```

Encoding for the Single-precision variant
(FEAT_FP) Applies when `(ftype == 00)`

```
FNEG <Sd>, <Sn>
```

Encoding for the Double-precision variant
(FEAT_FP) Applies when `(ftype == 01)`

```
FNEG <Dd>, <Dn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if ftype == '10' then EndOfDecode(Decode_UNDEF);
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 8 << UInt(ftype EOR '10');
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<**Sd**> Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Sn**> Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<**Dd**> Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Dn**> Is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(esize) operand = V[n, esize];

constant bits(esize) result = FPNeg(operand, FPCR);

V[d, esize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

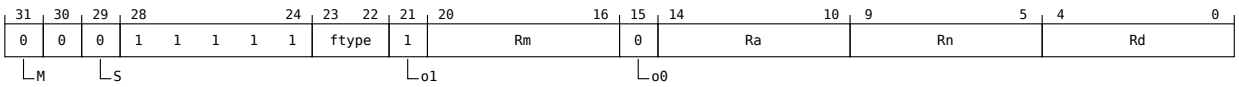
C7.2.134 FNMADD

Floating-point negated fused multiply-add (scalar)

This instruction multiplies the values of the first two SIMD&FP source registers, negates the product, subtracts the value of the third SIMD&FP source register, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Encoding for the Half-precision variant
(FEAT_FP16) Applies when `(ftype == 11)`

```
FNMADD <Hd>, <Hn>, <Hm>, <Ha>
```

Encoding for the Single-precision variant
(FEAT_FP) Applies when `(ftype == 00)`

```
FNMADD <Sd>, <Sn>, <Sm>, <Sa>
```

Encoding for the Double-precision variant
(FEAT_FP) Applies when `(ftype == 01)`

```
FNMADD <Dd>, <Dn>, <Dm>, <Da>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if ftype == '10' then EndOfDecode(Decode_UNDEF);
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer a = UInt(Ra);

constant integer esize = 8 << UInt(ftype EOR '10');
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the “Rn” field.

<**Hm**> Is the 16-bit name of the second SIMD&FP source register holding the multiplier, encoded in the “Rm” field.

<**Ha**> Is the 16-bit name of the third SIMD&FP source register holding the addend, encoded in the “Ra” field.

<**Sd**> Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Sn**> Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the “Rn” field.

<**Sm**> Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the “Rm” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**Sa**> Is the 32-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.

<**Dd**> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<**Dn**> Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.

<**Dm**> Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

<**Da**> Is the 64-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(esize) addend   = FPNeg(V[a, esize], FPCR);
constant bits(esize) operand1 = FPNeg(V[n, esize], FPCR);
constant bits(esize) operand2 = V[m, esize];

V[d, esize] = FPMulAdd(addend, operand1, operand2, FPCR);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

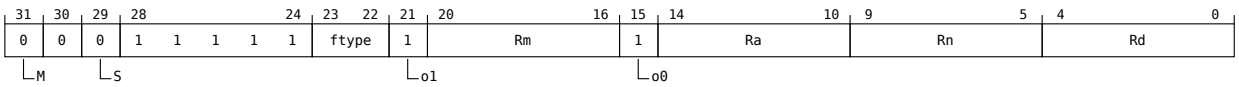
C7.2.135 FNMSUB

Floating-point negated fused multiply-subtract (scalar)

This instruction multiplies the values of the first two SIMD&FP source registers, subtracts the value of the third SIMD&FP source register, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Encoding for the Half-precision variant
(FEAT_FP16) Applies when `(ftype == 11)`

```
FNMSUB <Hd>, <Hn>, <Hm>, <Ha>
```

Encoding for the Single-precision variant
(FEAT_FP) Applies when `(ftype == 00)`

```
FNMSUB <Sd>, <Sn>, <Sm>, <Sa>
```

Encoding for the Double-precision variant
(FEAT_FP) Applies when `(ftype == 01)`

```
FNMSUB <Dd>, <Dn>, <Dm>, <Da>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if ftype == '10' then EndOfDecode(Decode_UNDEF);
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer a = UInt(Ra);

constant integer esize = 8 << UInt(ftype EOR '10');
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the “Rn” field.

<**Hm**> Is the 16-bit name of the second SIMD&FP source register holding the multiplier, encoded in the “Rm” field.

<**Ha**> Is the 16-bit name of the third SIMD&FP source register holding the minuend, encoded in the “Ra” field.

<**Sd**> Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Sn**> Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the “Rn” field.

<**Sm**> Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the “Rm” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**Sa**> Is the 32-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.

<**Dd**> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<**Dn**> Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.

<**Dm**> Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

<**Da**> Is the 64-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(esize) addend   = FPNeg(V[a, esize], FPCR);
constant bits(esize) operand1 = V[n, esize];
constant bits(esize) operand2 = V[m, esize];

V[d, esize] = FPMulAdd(addend, operand1, operand2, FPCR);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

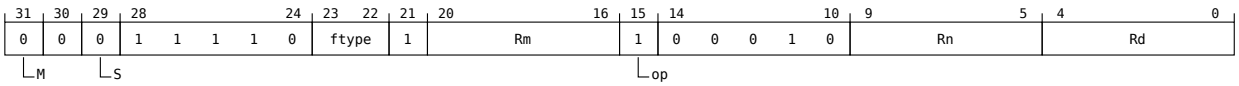
C7.2.136 FNMUL (scalar)

Floating-point multiply-negate (scalar)

This instruction multiplies the floating-point values of the two source SIMD&FP registers, and writes the negation of the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Encoding for the Half-precision variant
(FEAT_FP16) Applies when `(ftype == 11)`

```
FNMUL <Hd>, <Hn>, <Hm>
```

Encoding for the Single-precision variant
(FEAT_FP) Applies when `(ftype == 00)`

```
FNMUL <Sd>, <Sn>, <Sm>
```

Encoding for the Double-precision variant
(FEAT_FP) Applies when `(ftype == 01)`

```
FNMUL <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if ftype == '10' then EndOfDecode(Decode_UNDEF);
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 8 << UInt(ftype EOR '10');
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Hm**> Is the 16-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<**Sd**> Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Sn**> Is the 32-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Sm**> Is the 32-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<**Dd**> Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Dn**> Is the 64-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**Dm**> Is the 64-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(esize) operand1 = V[n, esize];
constant bits(esize) operand2 = V[m, esize];

constant bits(esize) result = FPMul(operand1, operand2, FPCR);
V[d, esize] = FPNeg(result, FPCR);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.137 FRECPE

Floating-point reciprocal estimate

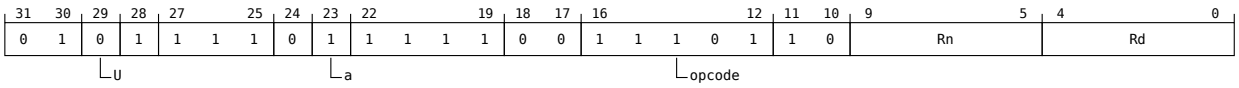
This instruction finds an approximate reciprocal estimate for each vector element in the source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision , Scalar single-precision and double-precision , Vector half-precision and Vector single-precision and double-precision

Scalar half-precision
(FEAT_AdvSIMD && FEAT_FP16)



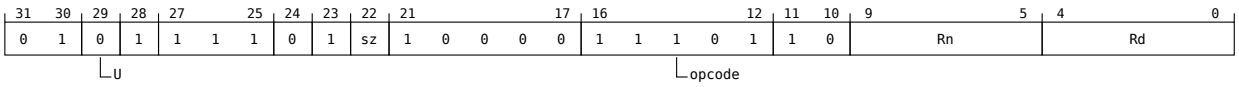
Encoding

```
FRECPE <Hd>, <Hn>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 16;
constant integer datasize = esize;
constant integer elements = 1;
```

Scalar single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

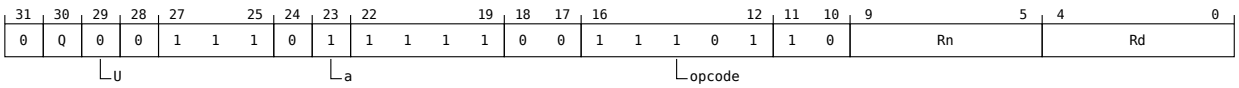
```
FRECPE <V><d>, <V><n>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 32 << UInt(sz);
constant integer datasize = esize;
constant integer elements = 1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Vector half-precision (FEAT_AdvSIMD && FEAT_FP16)



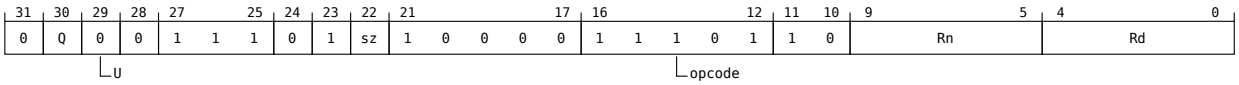
Encoding

```
FRECPE <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Vector single-precision and double-precision (FEAT_AdvSIMD)



Encoding

```
FRECPE <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<**V**> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<**d**> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**n**> Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> For the “Vector half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Vector single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];

bits(datasize) result;

for e = 0 to elements-1
    constant bits(esize) element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRecipEstimate(element, FPCR);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.138 FRECPS

Floating-point reciprocal step

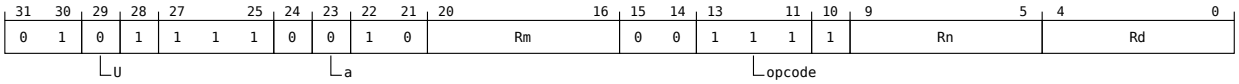
This instruction multiplies the corresponding floating-point values in the vectors of the two source SIMD&FP registers, subtracts each of the products from 2.0, places the resulting floating-point values in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision , Scalar single-precision and double-precision , Vector half-precision and Vector single-precision and double-precision

Scalar half-precision
(FEAT_AdvSIMD && FEAT_FP16)



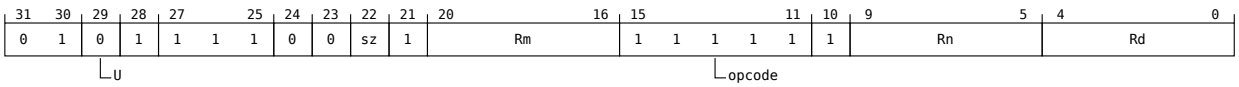
Encoding

```
FRECPS <Hd>, <Hn>, <Hm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 16;
constant integer datasize = esize;
constant integer elements = 1;
```

Scalar single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

```
FRECPS <V><d>, <V><n>, <V><m>
```

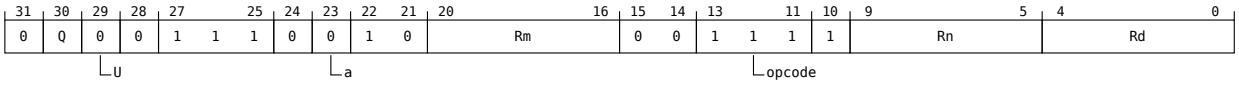
Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 32 << UInt(sz);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer datasize = esize;
constant integer elements = 1;
```

Vector half-precision (FEAT_AdvSIMD && FEAT_FP16)



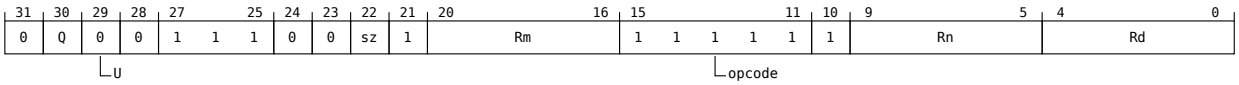
Encoding

```
FRECPS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Vector single-precision and double-precision (FEAT_AdvSIMD)



Encoding

```
FRECPS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Hm**> Is the 16-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<**V**> Is a width specifier, encoded in “sz”:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	<V>
0	S
1	D

<**d**> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<**n**> Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<**m**> Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> For the “Vector half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Vector single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];

bits(esize) element1;
bits(esize) element2;
bits(datasize) result;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPRecipStepFused(element1, element2, FPCR);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.139 FRECPX

Floating-point reciprocal exponent (scalar)

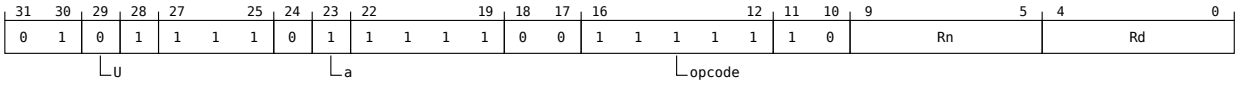
This instruction finds an approximate reciprocal exponent for the source SIMD&FP register and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(FEAT_AdvSIMD && FEAT_FP16)



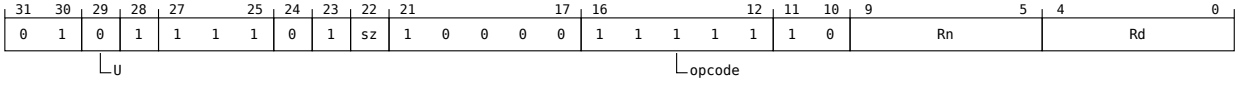
Encoding

```
FRECPX <Hd>, <Hn>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 16;
```

Single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

```
FRECPX <V><d>, <V><n>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 32 << UInt(sz);
```

Assembler Symbols

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<**Hn**> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<**V**> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**n**> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(esize) operand = V[n, esize];

bits(esize) result;

Elem[result, 0, esize] = FPRecpX(operand, FPCR);

V[d, esize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.140 FRINTA (vector)

Floating-point round to integral, to nearest with ties to away (vector)

This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round to Nearest with Ties to Away rounding mode, and writes the result to the SIMD&FP destination register.

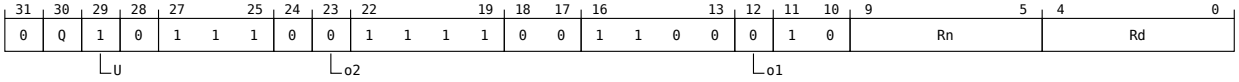
A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(FEAT_AdvSIMD && FEAT_FP16)



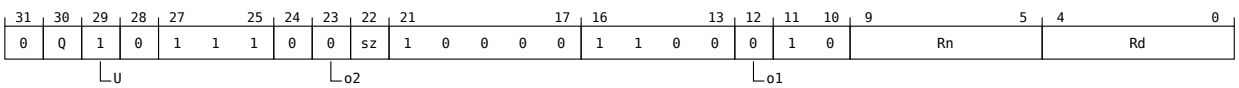
Encoding

```
FRINTA <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;

constant boolean exact = FALSE;
constant FPRounding rounding = FPRounding_TIEAWAY;
```

Single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

```
FRINTA <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;

constant boolean exact = FALSE;
constant FPRounding rounding = FPRounding_TIEAWAY;
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> For the "Half-precision" variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the "Single-precision and double-precision" variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundInt(element, FPCR, rounding, exact);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.141 FRINTA (scalar)

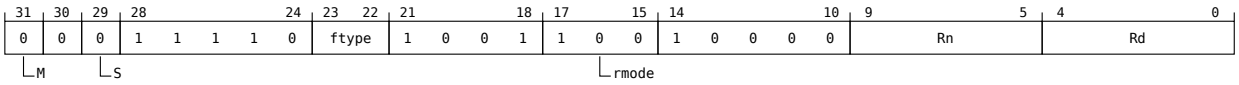
Floating-point round to integral, to nearest with ties to away (scalar)

This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round to Nearest with Ties to Away rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Encoding for the Half-precision variant
(FEAT_FP16) Applies when `(ftype == 11)`

```
FRINTA <Hd>, <Hn>
```

Encoding for the Single-precision variant
(FEAT_FP) Applies when `(ftype == 00)`

```
FRINTA <Sd>, <Sn>
```

Encoding for the Double-precision variant
(FEAT_FP) Applies when `(ftype == 01)`

```
FRINTA <Dd>, <Dn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if ftype == '10' then EndOfDecode(Decode_UNDEF);
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 8 << UInt(ftype EOR '10');
constant boolean exact = FALSE;
constant FPRounding rounding = FPRounding_TIEAWAY;
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<**Sd**> Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Sn**> Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<**Dd**> Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**Dn**> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(esize) operand = V[n, esize];

V[d, esize] = FPRoundInt(operand, FPCR, rounding, exact);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.142 FRINTI (vector)

Floating-point round to integral, using current rounding mode (vector)

This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD&FP destination register.

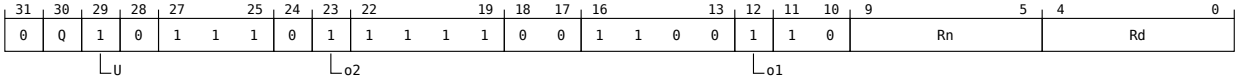
A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(FEAT_AdvSIMD && FEAT_FP16)



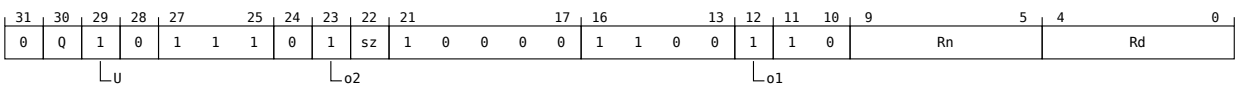
Encoding

```
FRINTI <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;

constant boolean exact = FALSE;
```

Single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

```
FRINTI <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer n = UInt(Rn);
constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;

constant boolean exact = FALSE;
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> For the "Half-precision" variant: is an arrangement specifier, encoded in "Q":

Q	<**T**>
0	4H
1	8H

For the "Single-precision and double-precision" variant: is an arrangement specifier, encoded in "sz:Q":

sz	**Q**	<**T**>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;
constant FPRounding rounding = FPRoundingMode(FPCR);

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundInt(element, FPCR, rounding, exact);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.143 FRINTI (scalar)

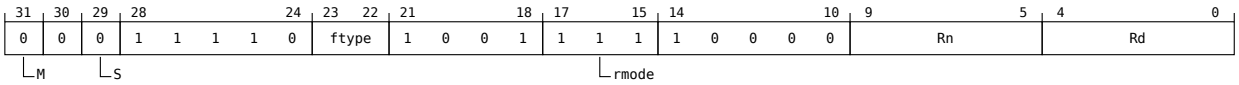
Floating-point round to integral, using current rounding mode (scalar)

This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Encoding for the Half-precision variant
(FEAT_FP16) Applies when `(ftype == 11)`

```
FRINTI <Hd>, <Hn>
```

Encoding for the Single-precision variant
(FEAT_FP) Applies when `(ftype == 00)`

```
FRINTI <Sd>, <Sn>
```

Encoding for the Double-precision variant
(FEAT_FP) Applies when `(ftype == 01)`

```
FRINTI <Dd>, <Dn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if ftype == '10' then EndOfDecode(Decode_UNDEF);
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 8 << UInt(ftype EOR '10');
constant boolean exact = FALSE;
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<**Sd**> Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Sn**> Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<**Dd**> Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**Dn**> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(esize) operand = V[n, esize];
constant FPRounding rounding = FPRoundingMode(FPCR);

V[d, esize] = FPRoundInt(operand, FPCR, rounding, exact);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.144 FRINTM (vector)

Floating-point round to integral, toward minus infinity (vector)

This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

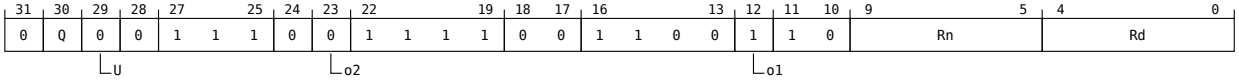
A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(FEAT_AdvSIMD && FEAT_FP16)



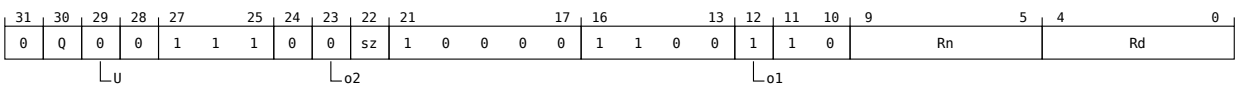
Encoding

```
FRINTM <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;

constant boolean exact = FALSE;
constant FPRounding rounding = FPRounding_NEGINF;
```

Single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

```
FRINTM <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;

constant boolean exact = FALSE;
constant FPRounding rounding = FPRounding_NEGINF;
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> For the "Half-precision" variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the "Single-precision and double-precision" variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundInt(element, FPCR, rounding, exact);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.145 FRINTM (scalar)

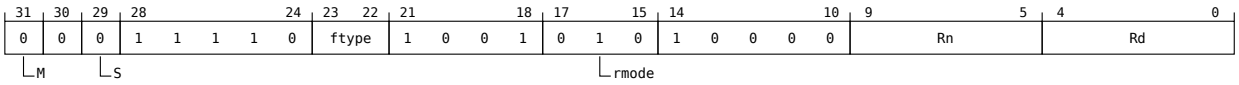
Floating-point round to integral, toward minus infinity (scalar)

This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Encoding for the Half-precision variant
(FEAT_FP16) Applies when `(ftype == 11)`

```
FRINTM <Hd>, <Hn>
```

Encoding for the Single-precision variant
(FEAT_FP) Applies when `(ftype == 00)`

```
FRINTM <Sd>, <Sn>
```

Encoding for the Double-precision variant
(FEAT_FP) Applies when `(ftype == 01)`

```
FRINTM <Dd>, <Dn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if ftype == '10' then EndOfDecode(Decode_UNDEF);
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 8 << UInt(ftype EOR '10');
constant boolean exact = FALSE;
constant FPRounding rounding = FPRounding_NEGINF;
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<**Sd**> Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Sn**> Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<**Dd**> Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(esize) operand = V[n, esize];

V[d, esize] = FPRoundInt(operand, FPCR, rounding, exact);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.146 FRINTN (vector)

Floating-point round to integral, to nearest with ties to even (vector)

This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round to Nearest rounding mode, and writes the result to the SIMD&FP destination register.

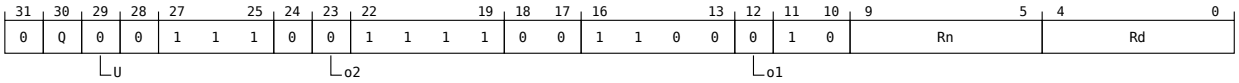
A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(FEAT_AdvSIMD && FEAT_FP16)



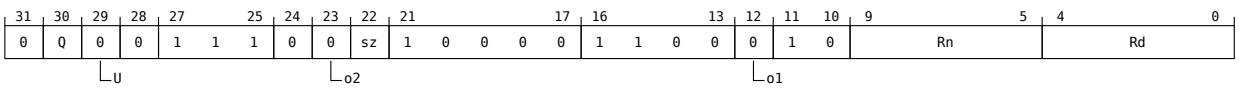
Encoding

```
FRINTN <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;

constant boolean exact = FALSE;
constant FPRounding rounding = FPRounding_TIEEVEN;
```

Single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

```
FRINTN <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer n = UInt(Rn);
constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;

constant boolean exact = FALSE;
constant FPRounding rounding = FPRounding_TIEEVEN;
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> For the "Half-precision" variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the "Single-precision and double-precision" variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundInt(element, FPCR, rounding, exact);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.147 FRINTN (scalar)

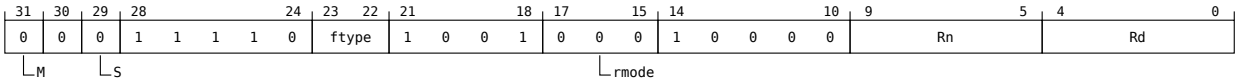
Floating-point round to integral, to nearest with ties to even (scalar)

This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round to Nearest rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Encoding for the Half-precision variant
(FEAT_FP16) Applies when `(ftype == 11)`

```
FRINTN <Hd>, <Hn>
```

Encoding for the Single-precision variant
(FEAT_FP) Applies when `(ftype == 00)`

```
FRINTN <Sd>, <Sn>
```

Encoding for the Double-precision variant
(FEAT_FP) Applies when `(ftype == 01)`

```
FRINTN <Dd>, <Dn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if ftype == '10' then EndOfDecode(Decode_UNDEF);
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 8 << UInt(ftype EOR '10');
constant boolean exact = FALSE;
constant FPRounding rounding = FPRounding_TIEEVEN;
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<**Sd**> Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Sn**> Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<**Dd**> Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Dn**> Is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(esize) operand = V[n, esize];

V[d, esize] = FPRoundInt(operand, FPCR, rounding, exact);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.148 FRINTP (vector)

Floating-point round to integral, toward plus infinity (vector)

This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

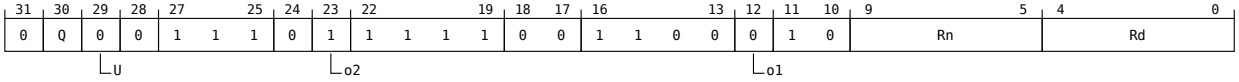
A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(FEAT_AdvSIMD && FEAT_FP16)



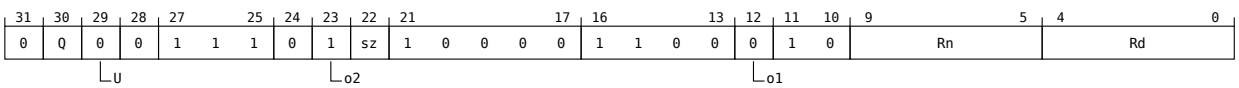
Encoding

```
FRINTP <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;

constant boolean exact = FALSE;
constant FPRounding rounding = FPRounding_POSINF;
```

Single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

```
FRINTP <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;

constant boolean exact = FALSE;
constant FPRounding rounding = FPRounding_POSINF;
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> For the "Half-precision" variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the "Single-precision and double-precision" variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundInt(element, FPCR, rounding, exact);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.149 FRINTP (scalar)

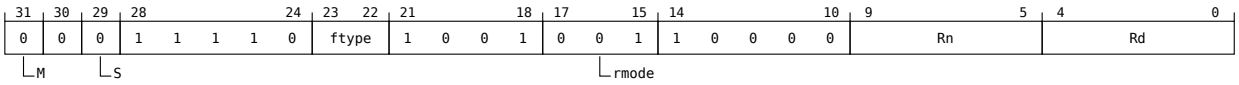
Floating-point round to integral, toward plus infinity (scalar)

This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Encoding for the Half-precision variant
(FEAT_FP16) Applies when `(ftype == 11)`

```
FRINTP <Hd>, <Hn>
```

Encoding for the Single-precision variant
(FEAT_FP) Applies when `(ftype == 00)`

```
FRINTP <Sd>, <Sn>
```

Encoding for the Double-precision variant
(FEAT_FP) Applies when `(ftype == 01)`

```
FRINTP <Dd>, <Dn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if ftype == '10' then EndOfDecode(Decode_UNDEF);
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 8 << UInt(ftype EOR '10');
constant boolean exact = FALSE;
constant FPRounding rounding = FPRounding_POSINF;
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<**Sd**> Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Sn**> Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<**Dd**> Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(esize) operand = V[n, esize];

V[d, esize] = FPRoundInt(operand, FPCR, rounding, exact);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.150 FRINTX (vector)

Floating-point round to integral exact, using current rounding mode (vector)

This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD&FP destination register.

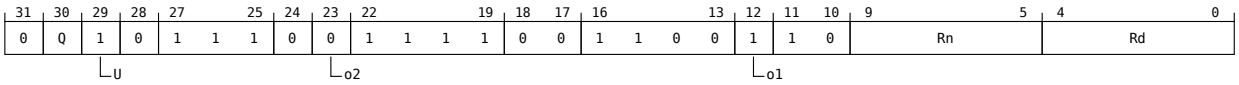
When a result value is not numerically equal to the corresponding input value, an Inexact exception is raised. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(FEAT_AdvSIMD && FEAT_FP16)



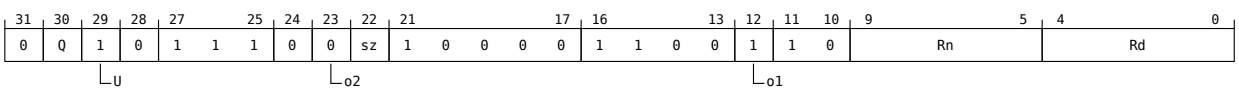
Encoding

```
FRINTX <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;

constant boolean exact = TRUE;
```

Single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

```
FRINTX <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;

constant boolean exact = TRUE;
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> For the "Half-precision" variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the "Single-precision and double-precision" variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;
constant FPRounding rounding = FPRoundingMode(FPCR);

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundInt(element, FPCR, rounding, exact);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.151 FRINTX (scalar)

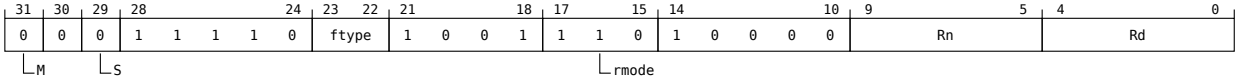
Floating-point round to integral exact, using current rounding mode (scalar)

This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD&FP destination register.

When the result value is not numerically equal to the input value, an Inexact exception is raised. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Encoding for the Half-precision variant
(FEAT_FP16) Applies when `(ftype == 11)`

```
FRINTX <Hd>, <Hn>
```

Encoding for the Single-precision variant
(FEAT_FP) Applies when `(ftype == 00)`

```
FRINTX <Sd>, <Sn>
```

Encoding for the Double-precision variant
(FEAT_FP) Applies when `(ftype == 01)`

```
FRINTX <Dd>, <Dn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if ftype == '10' then EndOfDecode(Decode_UNDEF);
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 8 << UInt(ftype EOR '10');
constant boolean exact = TRUE;
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<**Sd**> Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Sn**> Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<**Dd**> Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**Dn**> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(esize) operand = V[n, esize];
constant FPRounding rounding = FPRoundingMode(FPCR);

V[d, esize] = FPRoundInt(operand, FPCR, rounding, exact);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.152 FRINTZ (vector)

Floating-point round to integral, toward zero (vector)

This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

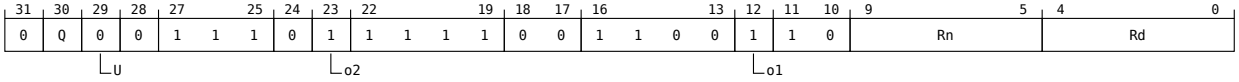
A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(FEAT_AdvSIMD && FEAT_FP16)



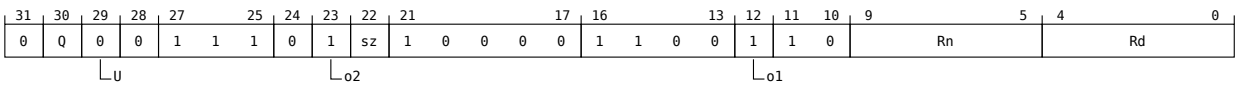
Encoding

```
FRINTZ <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;

constant boolean exact = FALSE;
constant FPRounding rounding = FPRounding_ZERO;
```

Single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

```
FRINTZ <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;

constant boolean exact = FALSE;
constant FPRounding rounding = FPRounding_ZERO;
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> For the "Half-precision" variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the "Single-precision and double-precision" variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundInt(element, FPCR, rounding, exact);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.153 FRINTZ (scalar)

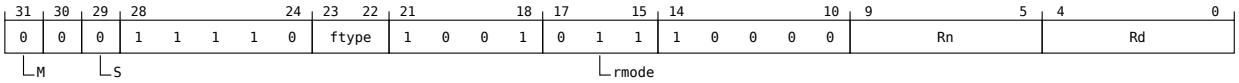
Floating-point round to integral, toward zero (scalar)

This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Encoding for the Half-precision variant
(FEAT_FP16) Applies when `(ftype == 11)`

```
FRINTZ <Hd>, <Hn>
```

Encoding for the Single-precision variant
(FEAT_FP) Applies when `(ftype == 00)`

```
FRINTZ <Sd>, <Sn>
```

Encoding for the Double-precision variant
(FEAT_FP) Applies when `(ftype == 01)`

```
FRINTZ <Dd>, <Dn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if ftype == '10' then EndOfDecode(Decode_UNDEF);
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 8 << UInt(ftype EOR '10');
constant boolean exact = FALSE;
constant FPRounding rounding = FPRounding_ZERO;
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<**Sd**> Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Sn**> Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<**Dd**> Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Dn**> Is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(esize) operand = V[n, esize];

V[d, esize] = FPRoundInt(operand, FPCR, rounding, exact);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.154 FRSQRTE

Floating-point reciprocal square root estimate

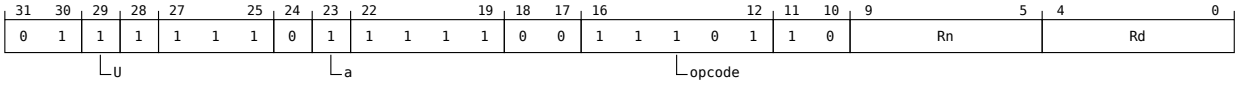
This instruction calculates an approximate square root for each vector element in the source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision , Scalar single-precision and double-precision , Vector half-precision and Vector single-precision and double-precision

Scalar half-precision
(FEAT_AdvSIMD && FEAT_FP16)



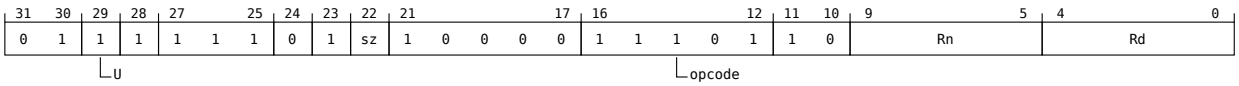
Encoding

```
FRSQRTE <Hd>, <Hn>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 16;
constant integer datasize = esize;
constant integer elements = 1;
```

Scalar single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

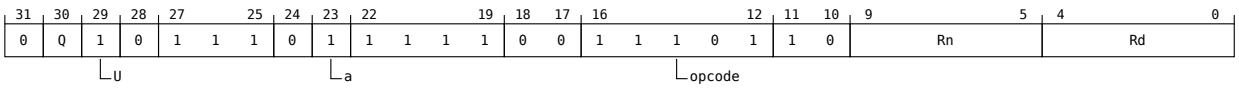
```
FRSQRTE <V><d>, <V><n>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 32 << UInt(sz);
constant integer datasize = esize;
constant integer elements = 1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Vector half-precision
(FEAT_AdvSIMD && FEAT_FP16)



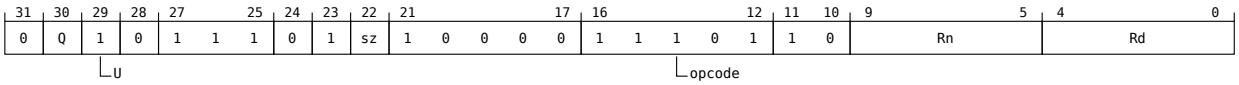
Encoding

```
FRSQRTE <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Vector single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

```
FRSQRTE <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<**V**> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<**d**> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**n**> Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> For the “Vector half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Vector single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
if elements == 1 then
    AArch64.CheckFPEnabled();
else
    AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];

bits(datasize) result;

for e = 0 to elements-1
    constant bits(esize) element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRSqrtEstimate(element, FPCR);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.155 FRSQRTS

Floating-point reciprocal square root step

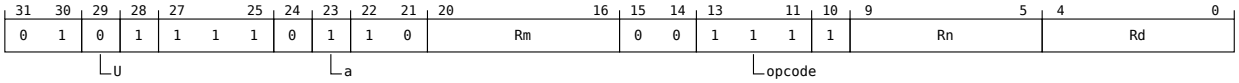
This instruction multiplies corresponding floating-point values in the vectors of the two source SIMD&FP registers, subtracts each of the products from 3.0, divides these results by 2.0, places the results into a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision , Scalar single-precision and double-precision , Vector half-precision and Vector single-precision and double-precision

Scalar half-precision
(FEAT_AdvSIMD && FEAT_FP16)



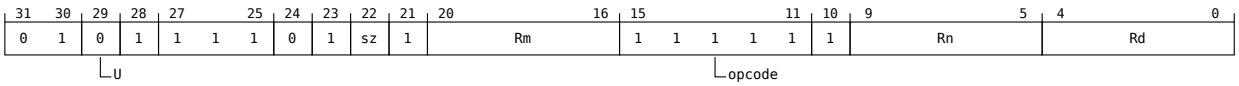
Encoding

```
FRSQRTS <Hd>, <Hn>, <Hm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 16;
constant integer datasize = esize;
constant integer elements = 1;
```

Scalar single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

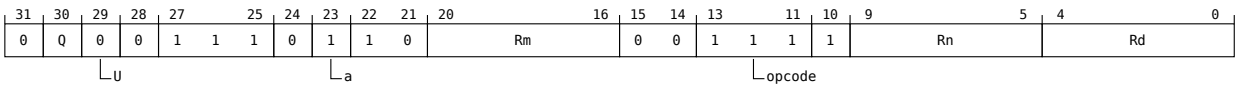
```
FRSQRTS <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 32 << UInt(sz);
constant integer datasize = esize;
constant integer elements = 1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Vector half-precision
(FEAT_AdvSIMD && FEAT_FP16)



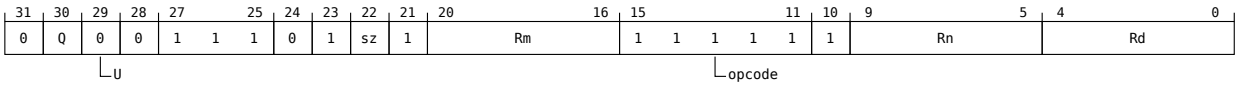
Encoding

```
FRSQRTS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Vector single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

```
FRSQRTS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Hm**> Is the 16-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<**V**> Is a width specifier, encoded in “sz”:

sz	<**V**>
0	S

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

sz	<V>
1	D

<**d**> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<**n**> Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<**m**> Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> For the “Vector half-precision” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “Vector single-precision and double-precision” variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];

bits(esize) element1;
bits(esize) element2;
bits(datasize) result;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPRSqrtStepFused(element1, element2, FPCR);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.156 FSQRT (vector)

Floating-point square root (vector)

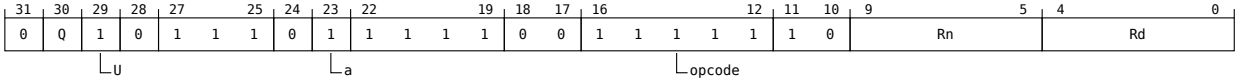
This instruction calculates the square root for each vector element in the source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(FEAT_AdvSIMD && FEAT_FP16)



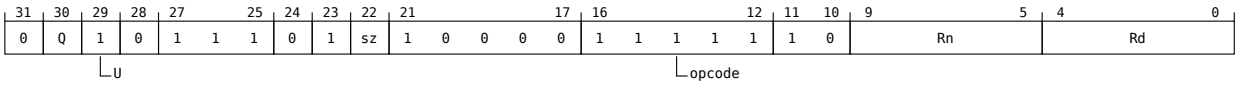
Encoding

```
FSQRT <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

```
FSQRT <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> For the "Half-precision" variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the "Single-precision and double-precision" variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(datasize) result;

for e = 0 to elements-1
    constant bits(esize) element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPSqrt(element, FPCR);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

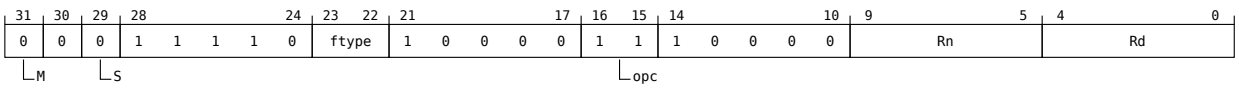
C7.2.157 FSQRT (scalar)

Floating-point square root (scalar)

This instruction calculates the square root of the value in the SIMD&FP source register and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Encoding for the Half-precision variant
(FEAT_FP16) Applies when `(ftype == 11)`

```
FSQRT <Hd>, <Hn>
```

Encoding for the Single-precision variant
(FEAT_FP) Applies when `(ftype == 00)`

```
FSQRT <Sd>, <Sn>
```

Encoding for the Double-precision variant
(FEAT_FP) Applies when `(ftype == 01)`

```
FSQRT <Dd>, <Dn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if ftype == '10' then EndOfDecode(Decode_UNDEF);
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 8 << UInt(ftype EOR '10');
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<**Sd**> Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Sn**> Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

<**Dd**> Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Dn**> Is the 64-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(esize) operand = V[n, esize];

constant bits(esize) result = FPSqrt(operand, FPCR);

V[d, esize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.158 FSUB (vector)

Floating-point subtract (vector)

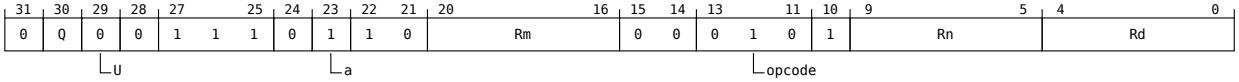
This instruction subtracts the elements in the vector in the second source SIMD&FP register, from the corresponding elements in the vector in the first source SIMD&FP register, places each result into elements of a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(FEAT_AdvSIMD && FEAT_FP16)



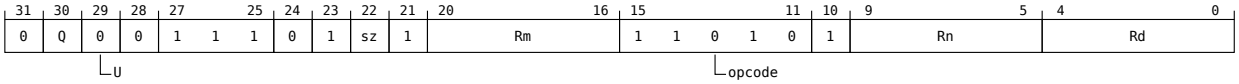
Encoding

```
FSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

```
FSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> For the "Half-precision" variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the "Single-precision and double-precision" variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];

bits(esize) element1;
bits(esize) element2;
bits(datasize) result;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPSub(element1, element2, FPCR);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

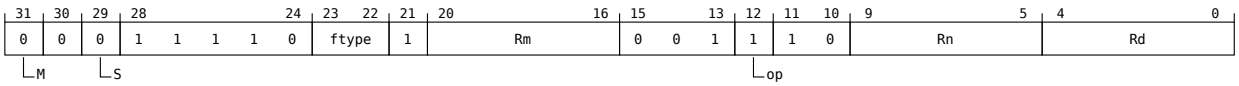
C7.2.159 FSUB (scalar)

Floating-point subtract (scalar)

This instruction subtracts the floating-point value of the second source SIMD&FP register from the floating-point value of the first source SIMD&FP register, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Encoding for the Half-precision variant
(FEAT_FP16) Applies when `(ftype == 11)`

```
FSUB <Hd>, <Hn>, <Hm>
```

Encoding for the Single-precision variant
(FEAT_FP) Applies when `(ftype == 00)`

```
FSUB <Sd>, <Sn>, <Sm>
```

Encoding for the Double-precision variant
(FEAT_FP) Applies when `(ftype == 01)`

```
FSUB <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if ftype == '10' then EndOfDecode(Decode_UNDEF);
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 8 << UInt(ftype EOR '10');
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Hm**> Is the 16-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<**Sd**> Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Sn**> Is the 32-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Sm**> Is the 32-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

<**Dd**> Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Dn**> Is the 64-bit name of the first SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**Dm**> Is the 64-bit name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(esize) operand1 = V[n, esize];
constant bits(esize) operand2 = V[m, esize];

V[d, esize] = FPSub(operand1, operand2, FPCR);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.160 INS (element)

Insert vector element from another vector element

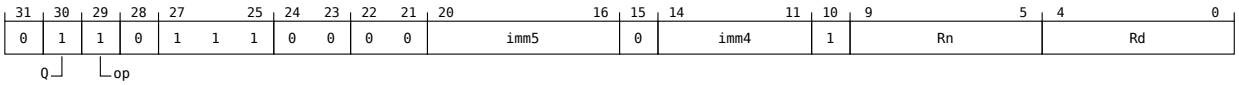
This instruction copies the vector element of the source SIMD&FP register to the specified vector element of the destination SIMD&FP register.

This instruction can insert data into individual elements within a SIMD&FP register without clearing the remaining bits to zero.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias MOV (element).

Advanced SIMD
(FEAT_AdvSIMD)



Encoding

```
INS <Vd>.<Ts>[<index1>], <Vn>.<Ts>[<index2>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if imm5 == 'x0000' then EndOfDecode(Decode_UNDEF);
constant integer size = LowestSetBitNZ(imm5<3:0>);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer dst_index = UInt(imm5<4:size+1>);
constant integer src_index = UInt(imm4<3:size>);
constant integer idxdsize = 64 << UInt(imm4<3>);
// imm4<size-1:0> is IGNORED
constant integer esize = 8 << size;
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Ts**> Is an element size specifier, encoded in “imm5”:

imm5	<Ts>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

<**index1**> Is the destination element index encoded in “imm5”:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

imm5	<index1>
x0000	RESERVED
xxxx1	UInt(imm5<4:1>)
xxx10	UInt(imm5<4:2>)
xx100	UInt(imm5<4:3>)
x1000	UInt(imm5<4>)

<**Vn**> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<**index2**> Is the source element index encoded in “imm5:imm4”:

imm5	<index2>
x0000	RESERVED
xxxx1	UInt(imm4)
xxx10	UInt(imm4<3:1>)
xx100	UInt(imm4<3:2>)
x1000	UInt(imm4<3>)

Unspecified bits in “imm4” are ignored but should be set to zero by an assembler.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(idxdsize) operand = V[n, idxdsize];
bits(128) result;

result = V[d, 128];
Elem[result, dst_index, esize] = Elem[operand, src_index, esize];
V[d, 128] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.161 INS (general)

Insert vector element from general-purpose register

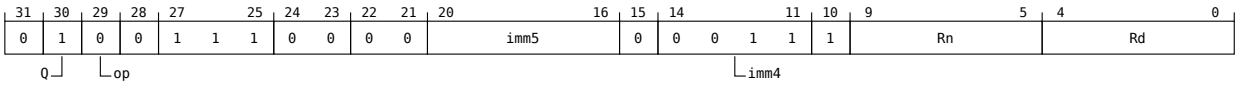
This instruction copies the contents of the source general-purpose register to the specified vector element in the destination SIMD&FP register.

This instruction can insert data into individual elements within a SIMD&FP register without clearing the remaining bits to zero.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias MOV (from general).

Advanced SIMD
(FEAT_AdvSIMD)



Encoding

```
INS <Vd>.<Ts>[<index>], <R><n>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if imm5 == 'x0000' then EndOfDecode(Decode_UNDEF);
constant integer size = LowestSetBitNZ(imm5<3:0>);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer index = UInt(imm5<4:size+1>);
constant integer esize = 8 << size;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ts> Is an element size specifier, encoded in “imm5”:

imm5	<Ts>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

<index> Is the element index encoded in “imm5”:

imm5	<index>
x0000	RESERVED

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

imm5	<index>
xxxx1	UInt(imm5<4:1>)
xxx10	UInt(imm5<4:2>)
xx100	UInt(imm5<4:3>)
x1000	UInt(imm5<4>)

<**R**> Is the width specifier for the general-purpose source register, encoded in “imm5”:

imm5	<R>
x0000	RESERVED
xxxx1	W
xxx10	W
xx100	W
x1000	X

<**n**> Is the number [0-30] of the general-purpose source register or ZR (31), encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(esize) element = X[n, esize];
bits(128) result = V[d, 128];

Elem[result, index, esize] = element;
V[d, 128] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.162 LD1 (multiple structures)

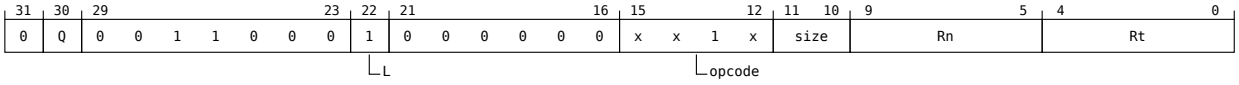
Load multiple single-element structures to one, two, three, or four registers

This instruction loads multiple single-element structures from memory and writes the result to one, two, three, or four SIMD&FP registers.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset
(FEAT_AdvSIMD)



Encoding for the One register variant Applies when `(opcode == 0111)`

```
LD1 { <Vt>.<T> }, [<Xn|SP>]
```

Encoding for the Two registers variant Applies when `(opcode == 1010)`

```
LD1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]
```

Encoding for the Three registers variant Applies when `(opcode == 0110)`

```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]
```

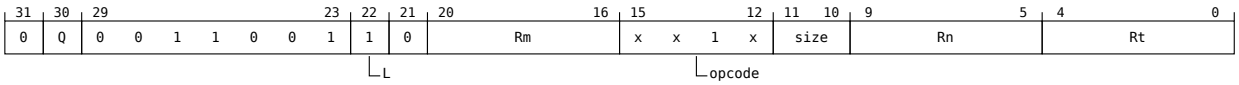
Encoding for the Four registers variant Applies when `(opcode == 0010)`

```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = integer UNKNOWN;
constant boolean wback = FALSE;
constant boolean nontemporal = FALSE;
```

Post-index
(FEAT_AdvSIMD)



Encoding for the One register, immediate offset variant Applies when `(Rm == 11111 && opcode == 0111)`

```
LD1 { <Vt>.<T> }, [<Xn|SP>], <imm>
```

Encoding for the One register, register offset variant Applies when `(Rm != 11111 && opcode == 0111)`

```
LD1 { <Vt>.<T> }, [<Xn|SP>], <Xm>
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Two registers, immediate offset variant Applies when `(Rm == 11111 && opcode == 1010)`

```
LD1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>
```

Encoding for the Two registers, register offset variant Applies when `(Rm != 11111 && opcode == 1010)`

```
LD1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>
```

Encoding for the Three registers, immediate offset variant Applies when `(Rm == 11111 && opcode == 0110)`

```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>
```

Encoding for the Three registers, register offset variant Applies when `(Rm != 11111 && opcode == 0110)`

```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>
```

Encoding for the Four registers, immediate offset variant Applies when `(Rm == 11111 && opcode == 0010)`

```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>
```

Encoding for the Four registers, register offset variant Applies when `(Rm != 11111 && opcode == 0010)`

```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant boolean wback = TRUE;
constant boolean nontemporal = FALSE;
```

Assembler Symbols

<**Vt**> Is the name of the first or only SIMD&FP register to be transferred, encoded in the “Rt” field.

<**T**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	1D
11	1	2D

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<**Vt2**> Is the name of the second SIMD&FP register to be transferred, encoded as “Rt” plus 1 modulo 32.

<**Vt3**> Is the name of the third SIMD&FP register to be transferred, encoded as “Rt” plus 2 modulo 32.

<**Vt4**> Is the name of the fourth SIMD&FP register to be transferred, encoded as “Rt” plus 3 modulo 32.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<imm> For the “One register, immediate offset” variant: is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#8
1	#16

For the “Two registers, immediate offset” variant: is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#16
1	#32

For the “Three registers, immediate offset” variant: is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#24
1	#48

For the “Four registers, immediate offset” variant: is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#32
1	#64

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

Shared Decode

```
constant integer datasize = 64 << UInt(Q);
constant integer esize = 8 << UInt(size);
constant integer elements = datasize DIV esize;

integer rpt;               // number of iterations
constant integer selem = 1; // structure elements

case opcode of
    when '0010' rpt = 4;    // LD/ST1 (4 registers)
    when '0110' rpt = 3;    // LD/ST1 (3 registers)
    when '1010' rpt = 2;    // LD/ST1 (2 registers)
    when '0111' rpt = 1;    // LD/ST1 (1 register)
    otherwise EndOfDecode(Decode_UNDEF);
```

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(64) address;
bits(64) eaddr;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_LOAD, nontemporal, privileged);
if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

offs = Zeros(64);
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt, datasize];
            eaddr = AddressIncrement(address, offs, accdesc);
            Elem[rval, e, esize] = Mem[eaddr, ebytes, accdesc];
            V[tt, datasize] = rval;
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;
if wback then
    if m != 31 then
        offs = X[m, 64];
    address = AddressAdd(address, offs, accdesc);
    if n == 31 then
        SP[64] = address;
    else
        X[n, 64] = address;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.163 LD1 (single structure)

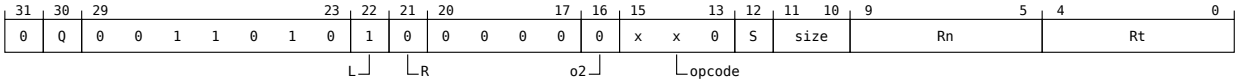
Load one single-element structure to one lane of one register

This instruction loads a single-element structure from memory and writes the result to the specified lane of the SIMD&FP register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset
(FEAT_AdvSIMD)



Encoding for the 8-bit variant Applies when `(opcode == 000)`

```
LD1 { <Vt>.B }[<index>], [<Xn|SP>]
```

Encoding for the 16-bit variant Applies when `(opcode == 010 && size == x0)`

```
LD1 { <Vt>.H }[<index>], [<Xn|SP>]
```

Encoding for the 32-bit variant Applies when `(opcode == 100 && size == 00)`

```
LD1 { <Vt>.S }[<index>], [<Xn|SP>]
```

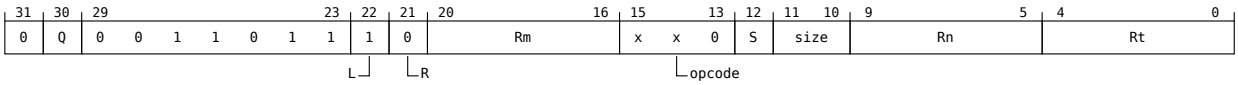
Encoding for the 64-bit variant Applies when `(opcode == 100 && S == 0 && size == 01)`

```
LD1 { <Vt>.D }[<index>], [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = integer UNKNOWN;
constant boolean wback = FALSE;
constant boolean nontemporal = FALSE;
```

Post-index
(FEAT_AdvSIMD)



Encoding for the 8-bit, immediate offset variant Applies when `(Rm == 11111 && opcode == 000)`

```
LD1 { <Vt>.B }[<index>], [<Xn|SP>], #1
```

Encoding for the 8-bit, register offset variant Applies when `(Rm != 11111 && opcode == 000)`

```
LD1 { <Vt>.B }[<index>], [<Xn|SP>], <Xm>
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 64-bit, immediate offset variant Applies when `(Rm == 11111 && opcode == 100 && S == 0 && size == 01)`

```
LD1 { <Vt>.D }[<index>], [<Xn|SP>], #8
```

Encoding for the 64-bit, register offset variant Applies when `(Rm != 11111 && opcode == 100 && S == 0 && size == 01)`

```
LD1 { <Vt>.D }[<index>], [<Xn|SP>], <Xm>
```

Encoding for the 16-bit, immediate offset variant Applies when `(Rm == 11111 && opcode == 010 && size == x0)`

```
LD1 { <Vt>.H }[<index>], [<Xn|SP>], #2
```

Encoding for the 16-bit, register offset variant Applies when `(Rm != 11111 && opcode == 010 && size == x0)`

```
LD1 { <Vt>.H }[<index>], [<Xn|SP>], <Xm>
```

Encoding for the 32-bit, immediate offset variant Applies when `(Rm == 11111 && opcode == 100 && size == 00)`

```
LD1 { <Vt>.S }[<index>], [<Xn|SP>], #4
```

Encoding for the 32-bit, register offset variant Applies when `(Rm != 11111 && opcode == 100 && size == 00)`

```
LD1 { <Vt>.S }[<index>], [<Xn|SP>], <Xm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant boolean wback = TRUE;
constant boolean nontemporal = FALSE;
```

Assembler Symbols

<**Vt**> Is the name of the first or only SIMD&FP register to be transferred, encoded in the “Rt” field.

<**index**> For the “8-bit”, “8-bit, immediate offset”, and “8-bit, register offset” variants: is the element index, encoded in “Q:S:size”.

For the “16-bit”, “16-bit, immediate offset”, and “16-bit, register offset” variants: is the element index, encoded in “Q:S:size<1>”.

For the “32-bit”, “32-bit, immediate offset”, and “32-bit, register offset” variants: is the element index, encoded in “Q:S”.

For the “64-bit”, “64-bit, immediate offset”, and “64-bit, register offset” variants: is the element index, encoded in “Q”.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<**Xm**> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

Shared Decode

```
bits(2) scale = opcode<2:1>;
constant integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when '11'
        // load and replicate
        if L == '0' || S == '1' then EndOfDecode(Decode_UNDEF);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        scale = size;
        replicate = TRUE;
    when '00'
        index = UInt(Q:S:size);       // B[0-15]
    when '01'
        if size<0> == '1' then EndOfDecode(Decode_UNDEF);
        index = UInt(Q:S:size<1>);    // H[0-7]
    when '10'
        if size<1> == '1' then EndOfDecode(Decode_UNDEF);
        if size<0> == '0' then
            index = UInt(Q:S);        // S[0-3]
        else
            if S == '1' then EndOfDecode(Decode_UNDEF);
            index = UInt(Q);          // D[0-1]
            scale = '11';

constant integer datasize = 64 << UInt(Q);
constant integer esize = 8 << UInt(scale);
```

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(64) address;
bits(64) eaddr;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_LOAD, nontemporal, privileged);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

offs = Zeros(64);

if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        eaddr = AddressIncrement(address, offs, accdesc);
        element = Mem[eaddr, ebytes, accdesc];
        // replicate to fill 128- or 64-bit register
        V[t, datasize] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t, 128];
        eaddr = AddressIncrement(address, offs, accdesc);
        Elem[rval, index, esize] = Mem[eaddr, ebytes, accdesc];
        V[t, 128] = rval;
        offs = offs + ebytes;
        t = ( t + 1 ) MOD 32;
if wback then
    if m != 31 then
        offs = X[m, 64];
    address = AddressAdd(address, offs, accdesc);
    if n == 31 then
        SP[64] = address;
    else
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        X[n, 64] = address;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.164 LD1R

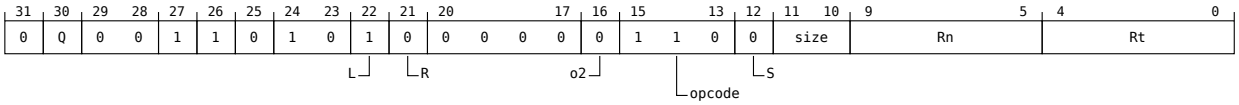
Load one single-element structure and replicate to all lanes (of one register)

This instruction loads a single-element structure from memory and replicates the structure to all the lanes of the SIMD&FP register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset
(FEAT_AdvSIMD)



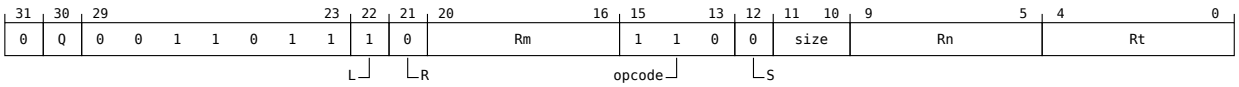
Encoding

```
LD1R { <Vt>.<T> }, [<Xn|SP>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = integer UNKNOWN;
constant boolean wback = FALSE;
constant boolean nontemporal = FALSE;
```

Post-index
(FEAT_AdvSIMD)



Encoding for the Immediate offset variant Applies when `(Rm == 11111)`

```
LD1R { <Vt>.<T> }, [<Xn|SP>], <imm>
```

Encoding for the Register offset variant Applies when `(Rm != 11111)`

```
LD1R { <Vt>.<T> }, [<Xn|SP>], <Xm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant boolean wback = TRUE;
constant boolean nontemporal = FALSE;
```

Assembler Symbols

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**Vt**> Is the name of the first or only SIMD&FP register to be transferred, encoded in the “Rt” field.

<**T**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	1D
11	1	2D

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<**imm**> Is the post-index immediate offset, encoded in “size”:

size	<imm>
00	#1
01	#2
10	#4
11	#8

<**Xm**> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

Shared Decode

```
bits(2) scale = opcode<2:1>;
constant integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when '11'
        // load and replicate
        if L == '0' || S == '1' then EndOfDecode(Decode_UNDEF);
        scale = size;
        replicate = TRUE;
    when '00'
        index = UInt(Q:S:size);       // B[0-15]
    when '01'
        if size<0> == '1' then EndOfDecode(Decode_UNDEF);
        index = UInt(Q:S:size<1>);    // H[0-7]
    when '10'
        if size<1> == '1' then EndOfDecode(Decode_UNDEF);
        if size<0> == '0' then
            index = UInt(Q:S);        // S[0-3]
        else
            if S == '1' then EndOfDecode(Decode_UNDEF);
            index = UInt(Q);          // D[0-1]
            scale = '11';

constant integer datasize = 64 << UInt(Q);
constant integer esize = 8 << UInt(scale);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(64) address;
bits(64) eaddr;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_LOAD, nontemporal, privileged);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

offs = Zeros(64);

if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        eaddr = AddressIncrement(address, offs, accdesc);
        element = Mem[eaddr, ebytes, accdesc];
        // replicate to fill 128- or 64-bit register
        V[t, datasize] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t, 128];
        eaddr = AddressIncrement(address, offs, accdesc);
        Elem[rval, index, esize] = Mem[eaddr, ebytes, accdesc];
        V[t, 128] = rval;
        offs = offs + ebytes;
        t = ( t + 1 ) MOD 32;
if wback then
    if m != 31 then
        offs = X[m, 64];
    address = AddressAdd(address, offs, accdesc);
    if n == 31 then
        SP[64] = address;
    else
        X[n, 64] = address;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.165 LD2 (multiple structures)

Load multiple 2-element structures to two registers

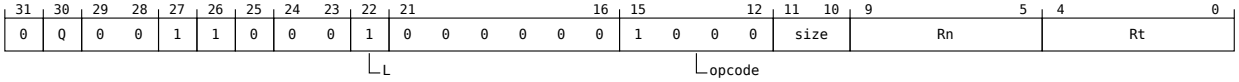
This instruction loads multiple 2-element structures from memory and writes the result to the two SIMD&FP registers, with de-interleaving.

For an example of de-interleaving, see LD3 (multiple structures).

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset
(FEAT_AdvSIMD)



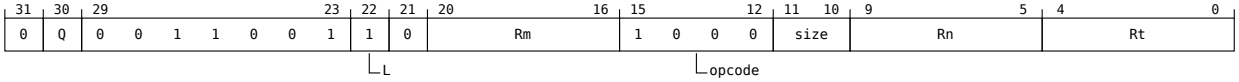
Encoding

```
LD2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = integer UNKNOWN;
constant boolean wback = FALSE;
constant boolean nontemporal = FALSE;
```

Post-index
(FEAT_AdvSIMD)



Encoding for the Immediate offset variant Applies when (Rm == 11111)

```
LD2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>
```

Encoding for the Register offset variant Applies when (Rm != 11111)

```
LD2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant boolean wback = TRUE;
constant boolean nontemporal = FALSE;
```

Assembler Symbols

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**Vt**> Is the name of the first or only SIMD&FP register to be transferred, encoded in the “Rt” field.

<**T**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<**Vt2**> Is the name of the second SIMD&FP register to be transferred, encoded as “Rt” plus 1 modulo 32.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<**imm**> Is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#16
1	#32

<**Xm**> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

Shared Decode

```
constant integer datasize = 64 << UInt(Q);
constant integer esize = 8 << UInt(size);
constant integer elements = datasize DIV esize;

constant integer rpt = 1;
constant integer selem = 2;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then EndOfDecode(Decode_UNDEF);
```

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(64) address;
bits(64) eaddr;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_LOAD, nontemporal, privileged);
if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
offs = Zeros(64);
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt, datasize];
            eaddr = AddressIncrement(address, offs, accdesc);
            Elem[rval, e, esize] = Mem[eaddr, ebytes, accdesc];
            V[tt, datasize] = rval;
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;
if wback then
    if m != 31 then
        offs = X[m, 64];
    address = AddressAdd(address, offs, accdesc);
    if n == 31 then
        SP[64] = address;
    else
        X[n, 64] = address;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.166 LD2 (single structure)

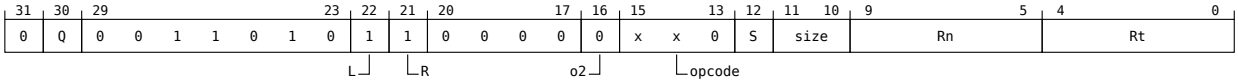
Load single 2-element structure to one lane of two registers

This instruction loads a 2-element structure from memory and writes the result to the corresponding elements of the two SIMD&FP registers without affecting the other bits of the registers.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset
(FEAT_AdvSIMD)



Encoding for the 8-bit variant Applies when `(opcode == 000)`

```
LD2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>]
```

Encoding for the 16-bit variant Applies when `(opcode == 010 && size == x0)`

```
LD2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>]
```

Encoding for the 32-bit variant Applies when `(opcode == 100 && size == 00)`

```
LD2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>]
```

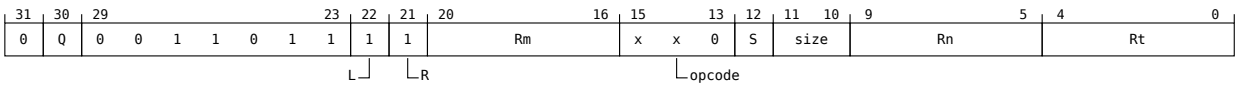
Encoding for the 64-bit variant Applies when `(opcode == 100 && S == 0 && size == 01)`

```
LD2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = integer UNKNOWN;
constant boolean wback = FALSE;
constant boolean nontemporal = FALSE;
```

Post-index
(FEAT_AdvSIMD)



Encoding for the 8-bit, immediate offset variant Applies when `(Rm == 11111 && opcode == 000)`

```
LD2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], #2
```

Encoding for the 8-bit, register offset variant Applies when `(Rm != 11111 && opcode == 000)`

```
LD2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], <Xm>
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 16-bit, immediate offset variant Applies when `(Rm == 11111 && opcode == 010 && size == x0)`

```
LD2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], #4
```

Encoding for the 16-bit, register offset variant Applies when `(Rm != 11111 && opcode == 010 && size == x0)`

```
LD2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], <Xm>
```

Encoding for the 32-bit, immediate offset variant Applies when `(Rm == 11111 && opcode == 100 && size == 00)`

```
LD2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>], #8
```

Encoding for the 32-bit, register offset variant Applies when `(Rm != 11111 && opcode == 100 && size == 00)`

```
LD2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>], <Xm>
```

Encoding for the 64-bit, immediate offset variant Applies when `(Rm == 11111 && opcode == 100 && S == 0 && size == 01)`

```
LD2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], #16
```

Encoding for the 64-bit, register offset variant Applies when `(Rm != 11111 && opcode == 100 && S == 0 && size == 01)`

```
LD2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], <Xm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant boolean wback = TRUE;
constant boolean nontemporal = FALSE;
```

Assembler Symbols

<**Vt**> Is the name of the first or only SIMD&FP register to be transferred, encoded in the “Rt” field.

<**Vt2**> Is the name of the second SIMD&FP register to be transferred, encoded as “Rt” plus 1 modulo 32.

<**index**> For the “8-bit”, “8-bit, immediate offset”, and “8-bit, register offset” variants: is the element index, encoded in “Q:S:size”.

For the “16-bit”, “16-bit, immediate offset”, and “16-bit, register offset” variants: is the element index, encoded in “Q:S:size<1>”.

For the “32-bit”, “32-bit, immediate offset”, and “32-bit, register offset” variants: is the element index, encoded in “Q:S”.

For the “64-bit”, “64-bit, immediate offset”, and “64-bit, register offset” variants: is the element index, encoded in “Q”.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<**Xm**> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

Shared Decode

```
bits(2) scale = opcode<2:1>;
constant integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when '11'
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        // load and replicate
        if L == '0' || S == '1' then EndOfDecode(Decode_UNDEF);
        scale = size;
        replicate = TRUE;
    when '00'
        index = UInt(Q:S:size);       // B[0-15]
    when '01'
        if size<0> == '1' then EndOfDecode(Decode_UNDEF);
        index = UInt(Q:S:size<1>);    // H[0-7]
    when '10'
        if size<1> == '1' then EndOfDecode(Decode_UNDEF);
        if size<0> == '0' then
            index = UInt(Q:S);        // S[0-3]
        else
            if S == '1' then EndOfDecode(Decode_UNDEF);
            index = UInt(Q);          // D[0-1]
            scale = '11';

constant integer datasize = 64 << UInt(Q);
constant integer esize = 8 << UInt(scale);
```

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(64) address;
bits(64) eaddr;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_LOAD, nontemporal, privileged);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

offs = Zeros(64);

if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        eaddr = AddressIncrement(address, offs, accdesc);
        element = Mem[eaddr, ebytes, accdesc];
        // replicate to fill 128- or 64-bit register
        V[t, datasize] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t, 128];
        eaddr = AddressIncrement(address, offs, accdesc);
        Elem[rval, index, esize] = Mem[eaddr, ebytes, accdesc];
        V[t, 128] = rval;
        offs = offs + ebytes;
        t = ( t + 1 ) MOD 32;
if wback then
    if m != 31 then
        offs = X[m, 64];
    address = AddressAdd(address, offs, accdesc);
    if n == 31 then
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        SP[64] = address;
    else
        X[n, 64] = address;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.167 LD2R

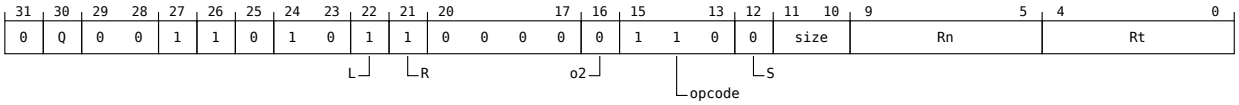
Load single 2-element structure and replicate to all lanes of two registers

This instruction loads a 2-element structure from memory and replicates the structure to all the lanes of the two SIMD&FP registers.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset
(FEAT_AdvSIMD)



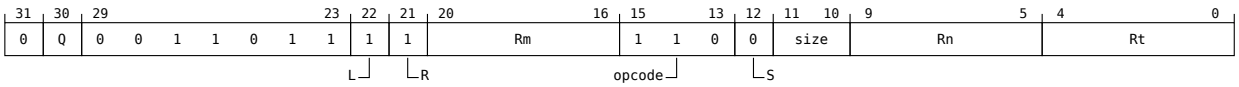
Encoding

```
LD2R { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = integer UNKNOWN;
constant boolean wback = FALSE;
constant boolean nontemporal = FALSE;
```

Post-index
(FEAT_AdvSIMD)



Encoding for the Immediate offset variant Applies when `(Rm == 11111)`

```
LD2R { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>
```

Encoding for the Register offset variant Applies when `(Rm != 11111)`

```
LD2R { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant boolean wback = TRUE;
constant boolean nontemporal = FALSE;
```

Assembler Symbols

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**Vt**> Is the name of the first or only SIMD&FP register to be transferred, encoded in the “Rt” field.

<**T**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	1D
11	1	2D

<**Vt2**> Is the name of the second SIMD&FP register to be transferred, encoded as “Rt” plus 1 modulo 32.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<**imm**> Is the post-index immediate offset, encoded in “size”:

size	<imm>
00	#2
01	#4
10	#8
11	#16

<**Xm**> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

Shared Decode

```
bits(2) scale = opcode<2:1>;
constant integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when '11'
        // load and replicate
        if L == '0' || S == '1' then EndOfDecode(Decode_UNDEF);
        scale = size;
        replicate = TRUE;
    when '00'
        index = UInt(Q:S:size);       // B[0-15]
    when '01'
        if size<0> == '1' then EndOfDecode(Decode_UNDEF);
        index = UInt(Q:S:size<1>);    // H[0-7]
    when '10'
        if size<1> == '1' then EndOfDecode(Decode_UNDEF);
        if size<0> == '0' then
            index = UInt(Q:S);        // S[0-3]
        else
            if S == '1' then EndOfDecode(Decode_UNDEF);
            index = UInt(Q);          // D[0-1]
            scale = '11';
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer datasize = 64 << UInt(Q);
constant integer esize = 8 << UInt(scale);
```

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(64) address;
bits(64) eaddr;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_LOAD, nontemporal, privileged);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

offs = Zeros(64);

if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        eaddr = AddressIncrement(address, offs, accdesc);
        element = Mem[eaddr, ebytes, accdesc];
        // replicate to fill 128- or 64-bit register
        V[t, datasize] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t, 128];
        eaddr = AddressIncrement(address, offs, accdesc);
        Elem[rval, index, esize] = Mem[eaddr, ebytes, accdesc];
        V[t, 128] = rval;
        offs = offs + ebytes;
        t = ( t + 1 ) MOD 32;
if wback then
    if m != 31 then
        offs = X[m, 64];
    address = AddressAdd(address, offs, accdesc);
    if n == 31 then
        SP[64] = address;
    else
        X[n, 64] = address;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

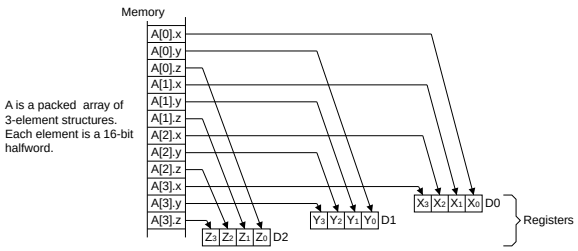
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.168 LD3 (multiple structures)

Load multiple 3-element structures to three registers

This instruction loads multiple 3-element structures from memory and writes the result to the three SIMD&FP registers, with de-interleaving.

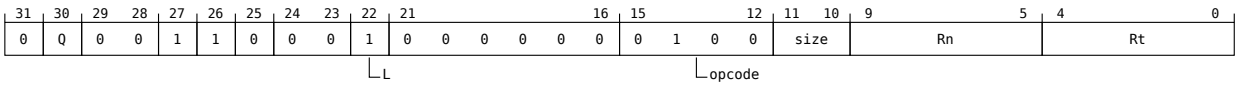
The following figure shows an example of the operation of de-interleaving of a LD3.16 (multiple 3-element structures) instruction:.



Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset
(FEAT_AdvSIMD)



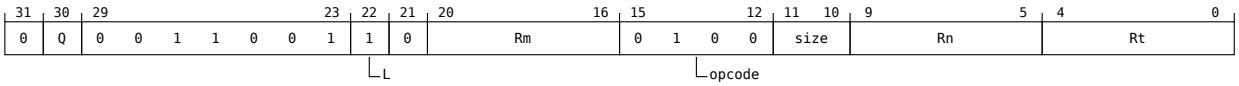
Encoding

```
LD3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = integer UNKNOWN;
constant boolean wback = FALSE;
constant boolean nontemporal = FALSE;
```

Post-index
(FEAT_AdvSIMD)



Encoding for the Immediate offset variant Applies when `(Rm == 11111)`

```
LD3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>
```

Encoding for the Register offset variant Applies when `(Rm != 11111)`

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
LD3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant boolean wback = TRUE;
constant boolean nontemporal = FALSE;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in "Q":

Q	<imm>
0	#24
1	#48

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
constant integer datasize = 64 << UInt(Q);
constant integer esize = 8 << UInt(size);
constant integer elements = datasize DIV esize;

constant integer rpt = 1;
constant integer selem = 3;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then EndOfDecode(Decode_UNDEF);
```

Operation

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
AArch64.CheckFPAdvSIMDEnabled();

bits(64) address;
bits(64) eaddr;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_LOAD, nontemporal, privileged);
if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

offs = Zeros(64);
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt, datasize];
            eaddr = AddressIncrement(address, offs, accdesc);
            Elem[rval, e, esize] = Mem[eaddr, ebytes, accdesc];
            V[tt, datasize] = rval;
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;
if wback then
    if m != 31 then
        offs = X[m, 64];
    address = AddressAdd(address, offs, accdesc);
    if n == 31 then
        SP[64] = address;
    else
        X[n, 64] = address;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.169 LD3 (single structure)

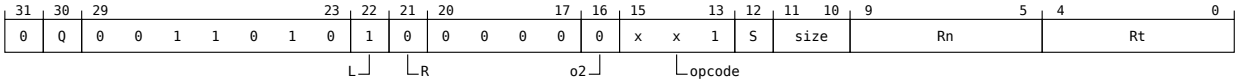
Load single 3-element structure to one lane of three registers

This instruction loads a 3-element structure from memory and writes the result to the corresponding elements of the three SIMD&FP registers without affecting the other bits of the registers.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset
(FEAT_AdvSIMD)



Encoding for the 8-bit variant Applies when `(opcode == 001)`

```
LD3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>]
```

Encoding for the 16-bit variant Applies when `(opcode == 011 && size == x0)`

```
LD3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>]
```

Encoding for the 32-bit variant Applies when `(opcode == 101 && size == 00)`

```
LD3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>]
```

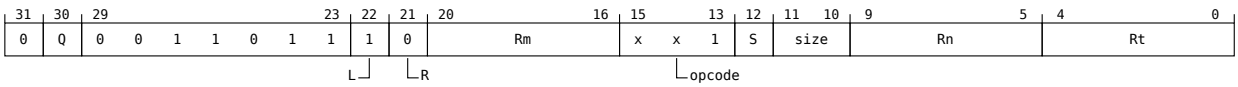
Encoding for the 64-bit variant Applies when `(opcode == 101 && S == 0 && size == 01)`

```
LD3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = integer UNKNOWN;
constant boolean wback = FALSE;
constant boolean nontemporal = FALSE;
```

Post-index
(FEAT_AdvSIMD)



Encoding for the 8-bit, immediate offset variant Applies when `(Rm == 11111 && opcode == 001)`

```
LD3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], #3
```

Encoding for the 8-bit, register offset variant Applies when `(Rm != 11111 && opcode == 001)`

```
LD3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], <Xm>
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 16-bit, immediate offset variant Applies when `(Rm == 11111 && opcode == 011 && size == x0)`

```
LD3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], #6
```

Encoding for the 16-bit, register offset variant Applies when `(Rm != 11111 && opcode == 011 && size == x0)`

```
LD3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], <Xm>
```

Encoding for the 32-bit, immediate offset variant Applies when `(Rm == 11111 && opcode == 101 && size == 00)`

```
LD3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], #12
```

Encoding for the 32-bit, register offset variant Applies when `(Rm != 11111 && opcode == 101 && size == 00)`

```
LD3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], <Xm>
```

Encoding for the 64-bit, immediate offset variant Applies when `(Rm == 11111 && opcode == 101 && S == 0 && size == 01)`

```
LD3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], #24
```

Encoding for the 64-bit, register offset variant Applies when `(Rm != 11111 && opcode == 101 && S == 0 && size == 01)`

```
LD3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], <Xm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant boolean wback = TRUE;
constant boolean nontemporal = FALSE;
```

Assembler Symbols

<**Vt**> Is the name of the first or only SIMD&FP register to be transferred, encoded in the “Rt” field.

<**Vt2**> Is the name of the second SIMD&FP register to be transferred, encoded as “Rt” plus 1 modulo 32.

<**Vt3**> Is the name of the third SIMD&FP register to be transferred, encoded as “Rt” plus 2 modulo 32.

<**index**> For the “8-bit”, “8-bit, immediate offset”, and “8-bit, register offset” variants: is the element index, encoded in “Q:S:size”.

For the “16-bit”, “16-bit, immediate offset”, and “16-bit, register offset” variants: is the element index, encoded in “Q:S:size<1>”.

For the “32-bit”, “32-bit, immediate offset”, and “32-bit, register offset” variants: is the element index, encoded in “Q:S”.

For the “64-bit”, “64-bit, immediate offset”, and “64-bit, register offset” variants: is the element index, encoded in “Q”.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<**Xm**> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

Shared Decode

```
bits(2) scale = opcode<2:1>;
constant integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
case scale of
    when '11'
        // load and replicate
        if L == '0' || S == '1' then EndOfDecode(Decode_UNDEF);
        scale = size;
        replicate = TRUE;
    when '00'
        index = UInt(Q:S:size);       // B[0-15]
    when '01'
        if size<0> == '1' then EndOfDecode(Decode_UNDEF);
        index = UInt(Q:S:size<1>);    // H[0-7]
    when '10'
        if size<1> == '1' then EndOfDecode(Decode_UNDEF);
        if size<0> == '0' then
            index = UInt(Q:S);        // S[0-3]
        else
            if S == '1' then EndOfDecode(Decode_UNDEF);
            index = UInt(Q);          // D[0-1]
            scale = '11';

constant integer datasize = 64 << UInt(Q);
constant integer esize = 8 << UInt(scale);
```

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(64) address;
bits(64) eaddr;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_LOAD, nontemporal, privileged);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

offs = Zeros(64);

if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        eaddr = AddressIncrement(address, offs, accdesc);
        element = Mem[eaddr, ebytes, accdesc];
        // replicate to fill 128- or 64-bit register
        V[t, datasize] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t, 128];
        eaddr = AddressIncrement(address, offs, accdesc);
        Elem[rval, index, esize] = Mem[eaddr, ebytes, accdesc];
        V[t, 128] = rval;
        offs = offs + ebytes;
        t = ( t + 1 ) MOD 32;
if wback then
    if m != 31 then
        offs = X[m, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
address = AddressAdd(address, offs, accdesc);
if n == 31 then
    SP[64] = address;
else
    X[n, 64] = address;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.170 LD3R

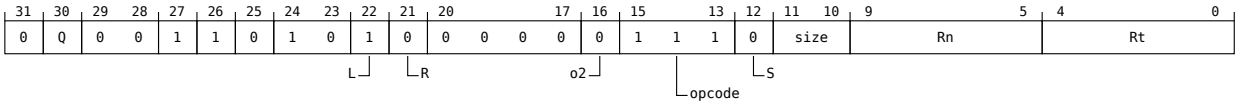
Load single 3-element structure and replicate to all lanes of three registers

This instruction loads a 3-element structure from memory and replicates the structure to all the lanes of the three SIMD&FP registers.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset
(FEAT_AdvSIMD)



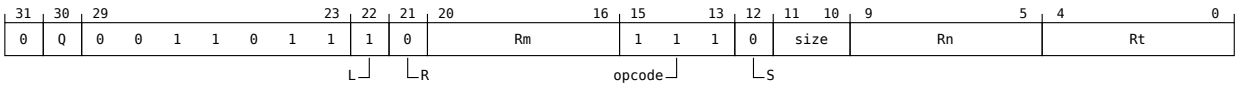
Encoding

```
LD3R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = integer UNKNOWN;
constant boolean wback = FALSE;
constant boolean nontemporal = FALSE;
```

Post-index
(FEAT_AdvSIMD)



Encoding for the Immediate offset variant Applies when `(Rm == 11111)`

```
LD3R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>
```

Encoding for the Register offset variant Applies when `(Rm != 11111)`

```
LD3R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant boolean wback = TRUE;
constant boolean nontemporal = FALSE;
```

Assembler Symbols

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**Vt**> Is the name of the first or only SIMD&FP register to be transferred, encoded in the “Rt” field.

<**T**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	1D
11	1	2D

<**Vt2**> Is the name of the second SIMD&FP register to be transferred, encoded as “Rt” plus 1 modulo 32.

<**Vt3**> Is the name of the third SIMD&FP register to be transferred, encoded as “Rt” plus 2 modulo 32.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<**imm**> Is the post-index immediate offset, encoded in “size”:

size	<imm>
00	#3
01	#6
10	#12
11	#24

<**Xm**> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

Shared Decode

```
bits(2) scale = opcode<2:1>;
constant integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when '11'
        // load and replicate
        if L == '0' || S == '1' then EndOfDecode(Decode_UNDEF);
        scale = size;
        replicate = TRUE;
    when '00'
        index = UInt(Q:S:size);       // B[0-15]
    when '01'
        if size<0> == '1' then EndOfDecode(Decode_UNDEF);
        index = UInt(Q:S:size<1>);    // H[0-7]
    when '10'
        if size<1> == '1' then EndOfDecode(Decode_UNDEF);
        if size<0> == '0' then
            index = UInt(Q:S);        // S[0-3]
        else
            if S == '1' then EndOfDecode(Decode_UNDEF);
            index = UInt(Q);          // D[0-1]
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            scale = '11';

constant integer datasize = 64 << UInt(Q);
constant integer esize = 8 << UInt(scale);
```

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(64) address;
bits(64) eaddr;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_LOAD, nontemporal, privileged);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

offs = Zeros(64);

if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        eaddr = AddressIncrement(address, offs, accdesc);
        element = Mem[eaddr, ebytes, accdesc];
        // replicate to fill 128- or 64-bit register
        V[t, datasize] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t, 128];
        eaddr = AddressIncrement(address, offs, accdesc);
        Elem[rval, index, esize] = Mem[eaddr, ebytes, accdesc];
        V[t, 128] = rval;
        offs = offs + ebytes;
        t = ( t + 1 ) MOD 32;
if wback then
    if m != 31 then
        offs = X[m, 64];
    address = AddressAdd(address, offs, accdesc);
    if n == 31 then
        SP[64] = address;
    else
        X[n, 64] = address;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.171 LD4 (multiple structures)

Load multiple 4-element structures to four registers

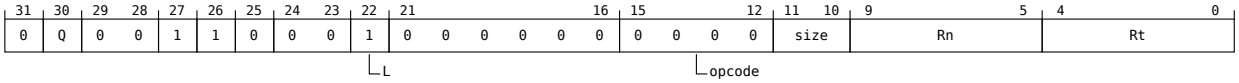
This instruction loads multiple 4-element structures from memory and writes the result to the four SIMD&FP registers, with de-interleaving.

For an example of de-interleaving, see LD3 (multiple structures).

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset
(FEAT_AdvSIMD)



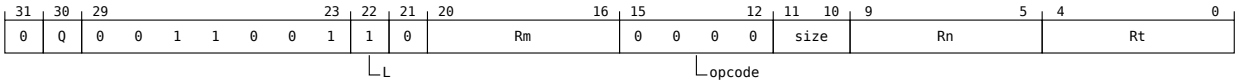
Encoding

```
LD4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = integer UNKNOWN;
constant boolean wback = FALSE;
constant boolean nontemporal = FALSE;
```

Post-index
(FEAT_AdvSIMD)



Encoding for the Immediate offset variant Applies when `(Rm == 11111)`

```
LD4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>
```

Encoding for the Register offset variant Applies when `(Rm != 11111)`

```
LD4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant boolean wback = TRUE;
constant boolean nontemporal = FALSE;
```

Assembler Symbols

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**Vt**> Is the name of the first or only SIMD&FP register to be transferred, encoded in the “Rt” field.

<**T**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<**Vt2**> Is the name of the second SIMD&FP register to be transferred, encoded as “Rt” plus 1 modulo 32.

<**Vt3**> Is the name of the third SIMD&FP register to be transferred, encoded as “Rt” plus 2 modulo 32.

<**Vt4**> Is the name of the fourth SIMD&FP register to be transferred, encoded as “Rt” plus 3 modulo 32.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<**imm**> Is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#32
1	#64

<**Xm**> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

Shared Decode

```
constant integer datasize = 64 << UInt(Q);
constant integer esize = 8 << UInt(size);
constant integer elements = datasize DIV esize;

constant integer rpt = 1;
constant integer selem = 4;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then EndOfDecode(Decode_UNDEF);
```

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(64) address;
bits(64) eaddr;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_LOAD, nontemporal, privileged);
if n == 31 then
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

offs = Zeros(64);
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt, datasize];
            eaddr = AddressIncrement(address, offs, accdesc);
            Elem[rval, e, esize] = Mem[eaddr, ebytes, accdesc];
            V[tt, datasize] = rval;
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;
if wback then
    if m != 31 then
        offs = X[m, 64];
    address = AddressAdd(address, offs, accdesc);
    if n == 31 then
        SP[64] = address;
    else
        X[n, 64] = address;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.172 LD4 (single structure)

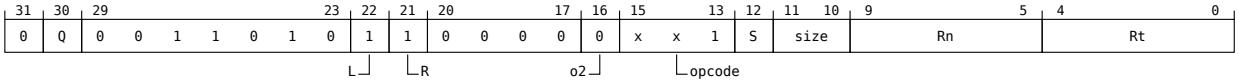
Load single 4-element structure to one lane of four registers

This instruction loads a 4-element structure from memory and writes the result to the corresponding elements of the four SIMD&FP registers without affecting the other bits of the registers.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset
(FEAT_AdvSIMD)



Encoding for the 8-bit variant Applies when `(opcode == 001)`

```
LD4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>]
```

Encoding for the 16-bit variant Applies when `(opcode == 011 && size == x0)`

```
LD4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>]
```

Encoding for the 32-bit variant Applies when `(opcode == 101 && size == 00)`

```
LD4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>]
```

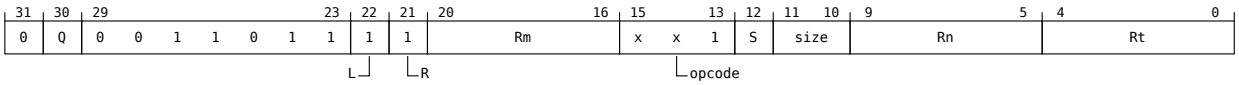
Encoding for the 64-bit variant Applies when `(opcode == 101 && S == 0 && size == 01)`

```
LD4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = integer UNKNOWN;
constant boolean wback = FALSE;
constant boolean nontemporal = FALSE;
```

Post-index
(FEAT_AdvSIMD)



Encoding for the 8-bit, immediate offset variant Applies when `(Rm == 11111 && opcode == 001)`

```
LD4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>], #4
```

Encoding for the 8-bit, register offset variant Applies when `(Rm != 11111 && opcode == 001)`

```
LD4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>], <Xm>
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 16-bit, immediate offset variant Applies when `(Rm == 11111 && opcode == 011 && size == x0)`

```
LD4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>], #8
```

Encoding for the 16-bit, register offset variant Applies when `(Rm != 11111 && opcode == 011 && size == x0)`

```
LD4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>], <Xm>
```

Encoding for the 32-bit, immediate offset variant Applies when `(Rm == 11111 && opcode == 101 && size == 00)`

```
LD4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>], #16
```

Encoding for the 32-bit, register offset variant Applies when `(Rm != 11111 && opcode == 101 && size == 00)`

```
LD4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>], <Xm>
```

Encoding for the 64-bit, immediate offset variant Applies when `(Rm == 11111 && opcode == 101 && S == 0 && size == 01)`

```
LD4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>], #32
```

Encoding for the 64-bit, register offset variant Applies when `(Rm != 11111 && opcode == 101 && S == 0 && size == 01)`

```
LD4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>], <Xm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant boolean wback = TRUE;
constant boolean nontemporal = FALSE;
```

Assembler Symbols

<**Vt**> Is the name of the first or only SIMD&FP register to be transferred, encoded in the “Rt” field.

<**Vt2**> Is the name of the second SIMD&FP register to be transferred, encoded as “Rt” plus 1 modulo 32.

<**Vt3**> Is the name of the third SIMD&FP register to be transferred, encoded as “Rt” plus 2 modulo 32.

<**Vt4**> Is the name of the fourth SIMD&FP register to be transferred, encoded as “Rt” plus 3 modulo 32.

<**index**> For the “8-bit”, “8-bit, immediate offset”, and “8-bit, register offset” variants: is the element index, encoded in “Q:S:size”.

For the “16-bit”, “16-bit, immediate offset”, and “16-bit, register offset” variants: is the element index, encoded in “Q:S:size<1>”.

For the “32-bit”, “32-bit, immediate offset”, and “32-bit, register offset” variants: is the element index, encoded in “Q:S”.

For the “64-bit”, “64-bit, immediate offset”, and “64-bit, register offset” variants: is the element index, encoded in “Q”.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<**Xm**> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

Shared Decode

```
bits(2) scale = opcode<2:1>;
constant integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
integer index;

case scale of
    when '11'
        // load and replicate
        if L == '0' || S == '1' then EndOfDecode(Decode_UNDEF);
        scale = size;
        replicate = TRUE;
    when '00'
        index = UInt(Q:S:size);       // B[0-15]
    when '01'
        if size<0> == '1' then EndOfDecode(Decode_UNDEF);
        index = UInt(Q:S:size<1>);    // H[0-7]
    when '10'
        if size<1> == '1' then EndOfDecode(Decode_UNDEF);
        if size<0> == '0' then
            index = UInt(Q:S);        // S[0-3]
        else
            if S == '1' then EndOfDecode(Decode_UNDEF);
            index = UInt(Q);          // D[0-1]
            scale = '11';

constant integer datasize = 64 << UInt(Q);
constant integer esize = 8 << UInt(scale);
```

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(64) address;
bits(64) eaddr;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_LOAD, nontemporal, privileged);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

offs = Zeros(64);

if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        eaddr = AddressIncrement(address, offs, accdesc);
        element = Mem[eaddr, ebytes, accdesc];
        // replicate to fill 128- or 64-bit register
        V[t, datasize] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t, 128];
        eaddr = AddressIncrement(address, offs, accdesc);
        Elem[rval, index, esize] = Mem[eaddr, ebytes, accdesc];
        V[t, 128] = rval;
        offs = offs + ebytes;
        t = ( t + 1 ) MOD 32;
if wback then
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if m != 31 then
    offs = X[m, 64];
address = AddressAdd(address, offs, accdesc);
if n == 31 then
    SP[64] = address;
else
    X[n, 64] = address;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.173 LD4R

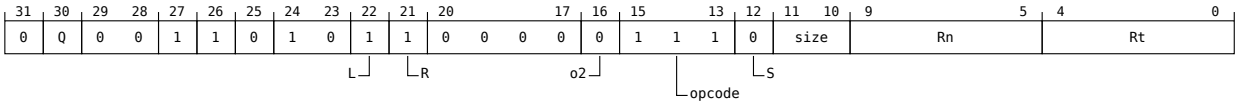
Load single 4-element structure and replicate to all lanes of four registers

This instruction loads a 4-element structure from memory and replicates the structure to all the lanes of the four SIMD&FP registers.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset
(FEAT_AdvSIMD)



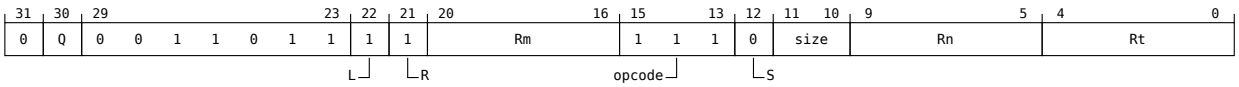
Encoding

```
LD4R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = integer UNKNOWN;
constant boolean wback = FALSE;
constant boolean nontemporal = FALSE;
```

Post-index
(FEAT_AdvSIMD)



Encoding for the Immediate offset variant Applies when `(Rm == 11111)`

```
LD4R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>
```

Encoding for the Register offset variant Applies when `(Rm != 11111)`

```
LD4R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant boolean wback = TRUE;
constant boolean nontemporal = FALSE;
```

Assembler Symbols

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**Vt**> Is the name of the first or only SIMD&FP register to be transferred, encoded in the “Rt” field.

<**T**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	1D
11	1	2D

<**Vt2**> Is the name of the second SIMD&FP register to be transferred, encoded as “Rt” plus 1 modulo 32.

<**Vt3**> Is the name of the third SIMD&FP register to be transferred, encoded as “Rt” plus 2 modulo 32.

<**Vt4**> Is the name of the fourth SIMD&FP register to be transferred, encoded as “Rt” plus 3 modulo 32.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<**imm**> Is the post-index immediate offset, encoded in “size”:

size	<imm>
00	#4
01	#8
10	#16
11	#32

<**Xm**> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

Shared Decode

```
bits(2) scale = opcode<2:1>;
constant integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when '11'
        // load and replicate
        if L == '0' || S == '1' then EndOfDecode(Decode_UNDEF);
        scale = size;
        replicate = TRUE;
    when '00'
        index = UInt(Q:S:size);       // B[0-15]
    when '01'
        if size<0> == '1' then EndOfDecode(Decode_UNDEF);
        index = UInt(Q:S:size<1>);    // H[0-7]
    when '10'
        if size<1> == '1' then EndOfDecode(Decode_UNDEF);
        if size<0> == '0' then
            index = UInt(Q:S);        // S[0-3]
        else
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            if S == '1' then EndOfDecode(Decode_UNDEF);
            index = UInt(Q);          // D[0-1]
            scale = '11';

constant integer datasize = 64 << UInt(Q);
constant integer esize = 8 << UInt(scale);
```

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(64) address;
bits(64) eaddr;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_LOAD, nontemporal, privileged);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

offs = Zeros(64);

if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        eaddr = AddressIncrement(address, offs, accdesc);
        element = Mem[eaddr, ebytes, accdesc];
        // replicate to fill 128- or 64-bit register
        V[t, datasize] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t, 128];
        eaddr = AddressIncrement(address, offs, accdesc);
        Elem[rval, index, esize] = Mem[eaddr, ebytes, accdesc];
        V[t, 128] = rval;
        offs = offs + ebytes;
        t = ( t + 1 ) MOD 32;
if wback then
    if m != 31 then
        offs = X[m, 64];
    address = AddressAdd(address, offs, accdesc);
    if n == 31 then
        SP[64] = address;
    else
        X[n, 64] = address;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.174 LDNP (SIMD&FP)

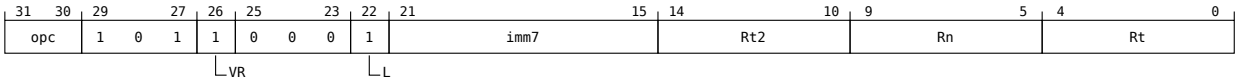
Load pair of SIMD&FP registers, with non-temporal hint

This instruction loads a pair of SIMD&FP registers from memory, issuing a hint to the memory system that the access is non-temporal. The address that is used for the load is calculated from a base register value and an optional immediate offset.

For information about non-temporal pair instructions, see Load/Store SIMD and Floating-point non-temporal pair.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Signed offset
(FEAT_FP)



Encoding for the 32-bit variant Applies when `(opc == 00)`

```
LDNP <St1>, <St2>, [<Xn|SP>{, #<imm>}]
```

Encoding for the 64-bit variant Applies when `(opc == 01)`

```
LDNP <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}]
```

Encoding for the 128-bit variant Applies when `(opc == 10)`

```
LDNP <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}]
```

Decode for all variants of this encoding

```
// Empty.
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDNP (SIMD&FP).

Assembler Symbols

<**St1**> Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the “Rt” field.

<**St2**> Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the “Rt2” field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<**imm**> For the “32-bit” variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the “imm7” field as <imm>/4.

For the “64-bit” variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the “imm7” field as <imm>/8.

For the “128-bit” variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the “imm7” field as <imm>/16.

<**Dt1**> Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the “Rt” field.

<**Dt2**> Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the “Rt2” field.

<**Qt1**> Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the “Rt” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Qt2> Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.

Shared Decode

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer t2 = UInt(Rt2);
constant integer n = UInt(Rn);
constant boolean nontemporal = TRUE;
constant integer scale = 2 + (UInt(opc));
constant integer datasize = 8 << scale;
constant bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean rt_unknown = FALSE;

if t == t2 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // Result is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
```

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
bits(64) address;
constant integer dbytes = datasize DIV 8;

constant boolean privileged = PSTATE.EL != EL0;
constant boolean ispair = TRUE;
constant AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_LOAD, nontemporal,
                                                       privileged, ispair);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

constant bits(2*datasize) data = Mem[address, 2*dbytes, accdesc];
if rt_unknown then
    V[t , datasize] = bits(datasize) UNKNOWN;
elsif BigEndian(accdesc.acctype) then
    V[t2, datasize] = data<(datasize-1):0>;
    V[t , datasize] = data<(2*datasize-1):datasize>;
else
    V[t , datasize] = data<(datasize-1):0>;
    V[t2, datasize] = data<(2*datasize-1):datasize>;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.175 LDP (SIMD&FP)

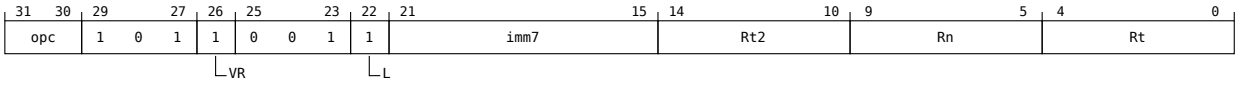
Load pair of SIMD&FP registers

This instruction loads a pair of SIMD&FP registers from memory. The address that is used for the load is calculated from a base register value and an optional immediate offset.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 3 classes: Post-index , Pre-index and Signed offset

Post-index
(FEAT_FP)



Encoding for the 32-bit variant Applies when `(opc == 00)`

```
LDP <St1>, <St2>, [<Xn|SP>], #<imm>
```

Encoding for the 64-bit variant Applies when `(opc == 01)`

```
LDP <Dt1>, <Dt2>, [<Xn|SP>], #<imm>
```

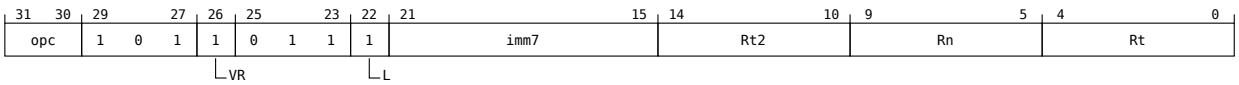
Encoding for the 128-bit variant Applies when `(opc == 10)`

```
LDP <Qt1>, <Qt2>, [<Xn|SP>], #<imm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);

constant boolean wback = TRUE;
constant boolean postindex = TRUE;
```

Pre-index
(FEAT_FP)



Encoding for the 32-bit variant Applies when `(opc == 00)`

```
LDP <St1>, <St2>, [<Xn|SP>, #<imm>]!
```

Encoding for the 64-bit variant Applies when `(opc == 01)`

```
LDP <Dt1>, <Dt2>, [<Xn|SP>, #<imm>]!
```

Encoding for the 128-bit variant Applies when `(opc == 10)`

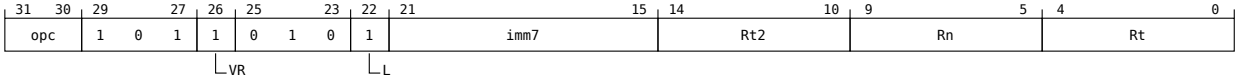
```
LDP <Qt1>, <Qt2>, [<Xn|SP>, #<imm>]!
```

Decode for all variants of this encoding

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);

constant boolean wback = TRUE;
constant boolean postindex = FALSE;
```

Signed offset
(FEAT_FP)



Encoding for the 32-bit variant Applies when `(opc == 00)`

```
LDP <St1>, <St2>, [<Xn|SP>{, #<imm>}]
```

Encoding for the 64-bit variant Applies when `(opc == 01)`

```
LDP <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}]
```

Encoding for the 128-bit variant Applies when `(opc == 10)`

```
LDP <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);

constant boolean wback = FALSE;
constant boolean postindex = FALSE;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDP (SIMD&FP).

Assembler Symbols

<**St1**> Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the “Rt” field.

<**St2**> Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the “Rt2” field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<**imm**> For the “Post-index 32-bit” and “Pre-index 32-bit” variants: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the “imm7” field as <imm>/4.

For the “Post-index 64-bit” and “Pre-index 64-bit” variants: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the “imm7” field as <imm>/8.

For the “Post-index 128-bit” and “Pre-index 128-bit” variants: is the signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, encoded in the “imm7” field as <imm>/16.

For the “Signed offset 32-bit” variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the “imm7” field as <imm>/4.

For the “Signed offset 64-bit” variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the “imm7” field as <imm>/8.

For the “Signed offset 128-bit” variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the “imm7” field as <imm>/16.

<**Dt1**> Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the “Rt” field.

<**Dt2**> Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the “Rt2” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**Qt1**> Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.

<**Qt2**> Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer t2 = UInt(Rt2);
constant integer n = UInt(Rn);
constant boolean nontemporal = FALSE;
constant integer scale = 2 + (UInt(opc));
constant integer datasize = 8 << scale;
constant bits(64) offset = LSL(SignExtend(imm7, 64), scale);

boolean rt_unknown = FALSE;

if t == t2 then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // Result is UNKNOWN
        when Constraint_UNDEF      EndOfDecode(Decode_UNDEF);
        when Constraint_NOP        EndOfDecode(Decode_NOP);
```

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
bits(64) address;
constant integer dbytes = datasize DIV 8;

constant boolean privileged = PSTATE.EL != EL0;
constant boolean ispair = TRUE;
constant AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_LOAD, nontemporal,
                                                       privileged, ispair);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

if !postindex then
    address = AddressAdd(address, offset, accdesc);

constant bits(2*datasize) data = Mem[address, 2*dbytes, accdesc];
if rt_unknown then
    V[t , datasize] = bits(datasize) UNKNOWN;
elsif BigEndian(accdesc.acctype) then
    V[t2, datasize] = data<(datasize-1):0>;
    V[t , datasize] = data<(2*datasize-1):datasize>;
else
    V[t , datasize] = data<(datasize-1):0>;
    V[t2, datasize] = data<(2*datasize-1):datasize>;

if wback then
    if postindex then
        address = AddressAdd(address, offset, accdesc);
    if n == 31 then
        SP[64] = address;
    else
        X[n, 64] = address;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.176 LDR (immediate, SIMD&FP)

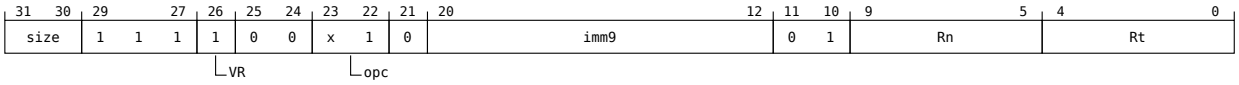
Load SIMD&FP register (immediate offset)

This instruction loads an element from memory, and writes the result as a scalar to the SIMD&FP register. The address that is used for the load is calculated from a base register value, a signed immediate offset, and an optional offset that is a multiple of the element size.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 3 classes: Post-index , Pre-index and Unsigned offset

Post-index
(FEAT_FP)



Encoding for the 8-bit variant Applies when `(size == 00 && opc == 01)`

```
LDR <Bt>, [<Xn|SP>], #<simm>
```

Encoding for the 16-bit variant Applies when `(size == 01 && opc == 01)`

```
LDR <Ht>, [<Xn|SP>], #<simm>
```

Encoding for the 32-bit variant Applies when `(size == 10 && opc == 01)`

```
LDR <St>, [<Xn|SP>], #<simm>
```

Encoding for the 64-bit variant Applies when `(size == 11 && opc == 01)`

```
LDR <Dt>, [<Xn|SP>], #<simm>
```

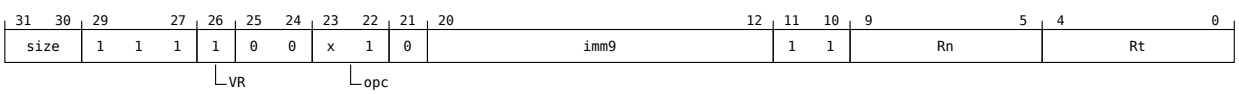
Encoding for the 128-bit variant Applies when `(size == 00 && opc == 11)`

```
LDR <Qt>, [<Xn|SP>], #<simm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if opc<1> == '1' && size != '00' then EndOfDecode(Decode_UNDEF);
constant integer scale = if opc<1> == '1' then 4 else UInt(size);
constant boolean wback = TRUE;
constant boolean postindex = TRUE;
constant bits(64) offset = SignExtend(imm9, 64);
```

Pre-index
(FEAT_FP)



Encoding for the 8-bit variant Applies when `(size == 00 && opc == 01)`

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
LDR <Bt>, [<Xn|SP>, #<simm>]!
```

Encoding for the 16-bit variant Applies when `(size == 01 && opc == 01)`

```
LDR <Ht>, [<Xn|SP>, #<simm>]!
```

Encoding for the 32-bit variant Applies when `(size == 10 && opc == 01)`

```
LDR <St>, [<Xn|SP>, #<simm>]!
```

Encoding for the 64-bit variant Applies when `(size == 11 && opc == 01)`

```
LDR <Dt>, [<Xn|SP>, #<simm>]!
```

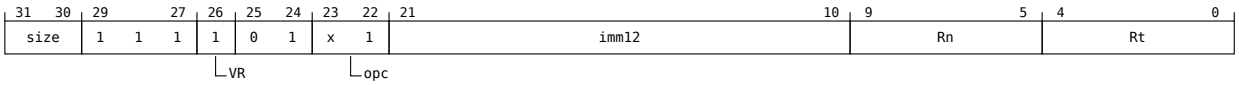
Encoding for the 128-bit variant Applies when `(size == 00 && opc == 11)`

```
LDR <Qt>, [<Xn|SP>, #<simm>]!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if opc<1> == '1' && size != '00' then EndOfDecode(Decode_UNDEF);
constant integer scale = if opc<1> == '1' then 4 else UInt(size);
constant boolean wback = TRUE;
constant boolean postindex = FALSE;
constant bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset
(FEAT_FP)



Encoding for the 8-bit variant Applies when `(size == 00 && opc == 01)`

```
LDR <Bt>, [<Xn|SP>{, #<pimm>}]
```

Encoding for the 16-bit variant Applies when `(size == 01 && opc == 01)`

```
LDR <Ht>, [<Xn|SP>{, #<pimm>}]
```

Encoding for the 32-bit variant Applies when `(size == 10 && opc == 01)`

```
LDR <St>, [<Xn|SP>{, #<pimm>}]
```

Encoding for the 64-bit variant Applies when `(size == 11 && opc == 01)`

```
LDR <Dt>, [<Xn|SP>{, #<pimm>}]
```

Encoding for the 128-bit variant Applies when `(size == 00 && opc == 11)`

```
LDR <Qt>, [<Xn|SP>{, #<pimm>}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if opc<1> == '1' && size != '00' then EndOfDecode(Decode_UNDEF);
constant integer scale = if opc<1> == '1' then 4 else UInt(size);
constant boolean wback = FALSE;
constant boolean postindex = FALSE;
constant bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<**Bt**> Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<**simm**> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.

<**Ht**> Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<**St**> Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<**Dt**> Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<**Qt**> Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<**pimm**> For the "8-bit" variant: is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For the "16-bit" variant: is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2.

For the "32-bit" variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4.

For the "64-bit" variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.

For the "128-bit" variant: is the optional positive immediate byte offset, a multiple of 16 in the range 0 to 65520, defaulting to 0 and encoded in the "imm12" field as <pimm>/16.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer datasize = 8 << scale;
constant boolean nontemporal = FALSE;
```

Operation

```
AArch64.CheckFPEnabled();
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_LOAD, nontemporal, privileged);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

if !postindex then
    address = AddressAdd(address, offset, accdesc);

V[t, datasize] = Mem[address, datasize DIV 8, accdesc];

if wback then
    if postindex then
        address = AddressAdd(address, offset, accdesc);
    if n == 31 then
        SP[64] = address;
    else
        X[n, 64] = address;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

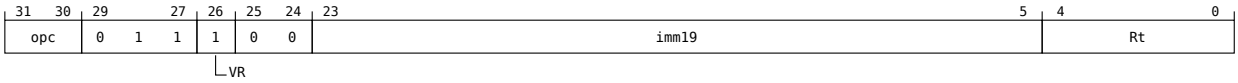
C7.2.177 LDR (literal, SIMD&FP)

Load SIMD&FP register (PC-relative literal)

This instruction loads a SIMD&FP register from memory. The address that is used for the load is calculated from the PC value and an immediate offset.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Literal
(FEAT_FP)



Encoding for the 32-bit variant Applies when `(opc == 00)`

```
LDR <St>, <label>
```

Encoding for the 64-bit variant Applies when `(opc == 01)`

```
LDR <Dt>, <label>
```

Encoding for the 128-bit variant Applies when `(opc == 10)`

```
LDR <Qt>, <label>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
if opc == '11' then EndOfDecode(Decode_UNDEF);
constant integer size = 4 << (UInt(opc));
constant boolean nontemporal = FALSE;
constant boolean tagchecked = FALSE;

constant bits(64) offset = SignExtend(imm19:'00', 64);
```

Assembler Symbols

<**St**> Is the 32-bit name of the SIMD&FP register to be loaded, encoded in the “Rt” field.

<**label**> Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as “imm19” times 4.

<**Dt**> Is the 64-bit name of the SIMD&FP register to be loaded, encoded in the “Rt” field.

<**Qt**> Is the 128-bit name of the SIMD&FP register to be loaded, encoded in the “Rt” field.

Operation

```
constant bits(64) address = PC64 + offset;
AArch64.CheckFPAdvSIMDEnabled();
constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_LOAD, nontemporal, privileged);
constant bits(size*8) data = Mem[address, size, accdesc];
V[t, size*8] = data;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

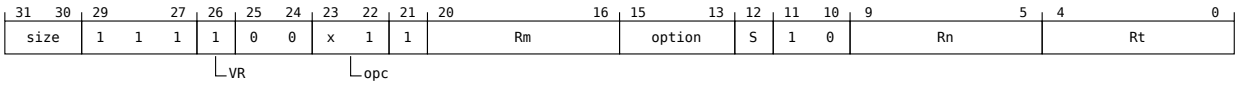
C7.2.178 LDR (register, SIMD&FP)

Load SIMD&FP register (register offset)

This instruction loads a SIMD&FP register from memory. The address that is used for the load is calculated from a base register value and an offset register value. The offset can be optionally shifted and extended.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

SIMD&FP registers
(FEAT_FP)



Encoding for the 8-bit with extended register offset variant Applies when `(size == 00 && opc == 01 && option != 011)`

```
LDR <Bt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

Encoding for the 8-bit with shifted register offset variant Applies when `(size == 00 && opc == 01 && option == 011)`

```
LDR <Bt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

Encoding for the 16-bit variant Applies when `(size == 01 && opc == 01)`

```
LDR <Ht>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Encoding for the 32-bit variant Applies when `(size == 10 && opc == 01)`

```
LDR <St>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Encoding for the 64-bit variant Applies when `(size == 11 && opc == 01)`

```
LDR <Dt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Encoding for the 128-bit variant Applies when `(size == 00 && opc == 11)`

```
LDR <Qt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if option<1> == '0' then EndOfDecode(Decode_UNDEF);             // sub-word index
if opc<1> == '1' && size != '00' then EndOfDecode(Decode_UNDEF);
constant integer scale = if opc<1> == '1' then 4 else UInt(size);
constant ExtendType extend_type = DecodeRegExtend(option);
constant integer shift = if S == '1' then scale else 0;
```

Assembler Symbols

<**Bt**> Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

<**Xn**|**SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<**Wm**> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the “Rm” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**Xm**> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<**extend**> For the "8-bit with extended register offset" variant: is the index extend specifier, encoded in "option":

option	<extend>
010	UXTW
110	SXTW
111	SXTX

For the "128-bit", "16-bit", "32-bit", and "64-bit" variants: is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted, encoded in "option":

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<**amount**> For the "8-bit with extended register offset" and "8-bit with shifted register offset" variants: is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.

For the "16-bit" variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

S	<amount>
0	#0
1	#1

For the "32-bit" variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

S	<amount>
0	#0
1	#2

For the "64-bit" variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

S	<amount>
0	#0
1	#3

For the "128-bit" variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

S	<amount>
0	#0
1	#4

<**Ht**> Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<**St**> Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<**Dt**> Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<**Qt**> Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 8 << scale;
constant boolean nontemporal = FALSE;
```

Operation

```
AArch64.CheckFPEnabled();
constant bits(64) offset = ExtendReg(m, extend_type, shift, 64);
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_LOAD, nontemporal, privileged);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

V[t, datasize] = Mem[address, datasize DIV 8, accdesc];
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

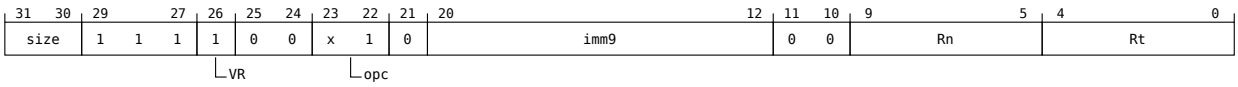
C7.2.179 LDUR (SIMD&FP)

Load SIMD&FP register (unscaled offset)

This instruction loads a SIMD&FP register from memory. The address that is used for the load is calculated from a base register value and an optional immediate offset.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Unscaled offset
(FEAT_FP)



Encoding for the 8-bit variant Applies when `(size == 00 && opc == 01)`

```
LDUR <Bt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 16-bit variant Applies when `(size == 01 && opc == 01)`

```
LDUR <Ht>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 32-bit variant Applies when `(size == 10 && opc == 01)`

```
LDUR <St>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 64-bit variant Applies when `(size == 11 && opc == 01)`

```
LDUR <Dt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 128-bit variant Applies when `(size == 00 && opc == 11)`

```
LDUR <Qt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if opc<1> == '1' && size != '00' then EndOfDecode(Decode_UNDEF);
constant integer scale = if opc<1> == '1' then 4 else UInt(size);
constant bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<**Bt**> Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<**simm**> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

<**Ht**> Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

<**St**> Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

<**Dt**> Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

<**Qt**> Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer datasize = 8 << scale;
constant boolean nontemporal = FALSE;
```

Operation

```
AArch64.CheckFPEnabled();
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_LOAD, nontemporal, privileged);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

V[t, datasize] = Mem[address, datasize DIV 8, accdesc];
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

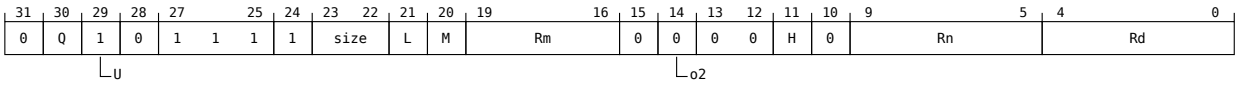
C7.2.180 MLA (by element)

Multiply-add to accumulator (vector, by element)

This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the results with the vector elements of the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector
(FEAT_AdvSIMD)



Encoding

```
MLA <Vd>.<T>, <Vn>.<T>, V<m>.<Ts>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer idxdsize = 64 << UInt(H);
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L);   Rmhi = M;
    otherwise EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rmhi:Rm);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**m**> Is the number of the second SIMD&FP source register, encoded in "size:M:Rm":

size	<**m**>
00	RESERVED
01	UInt('0':Rm)
10	UInt(M:Rm)
11	RESERVED

Restricted to 0-15 when element size <Ts> is H.

<**Ts**> Is an element size specifier, encoded in "size":

size	<**Ts**>
00	RESERVED
01	H
10	S
11	RESERVED

<**index**> Is the element index, encoded in "size:H:L:M":

size	<**index**>
00	RESERVED
01	UInt(H:L:M)
10	UInt(H:L)
11	RESERVED

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(idxdsize) operand2 = V[m, idxdsize];
constant bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) product;

element2 = UInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    product = (element1 * element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[operand3, e, esize] + product;
V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

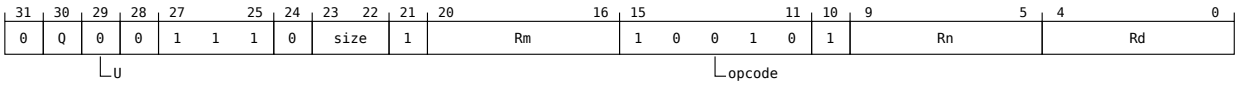
C7.2.181 MLA (vector)

Multiply-add to accumulator (vector)

This instruction multiplies corresponding elements in the vectors of the two source SIMD&FP registers, and accumulates the results with the vector elements of the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type
(FEAT_AdvSIMD)



Encoding

```
MLA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant bits(datasize) operand2 = V[m, datasize];
constant bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) product;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    product = (UInt(element1) * UInt(element2))<esize-1:0>;
    Elem[result, e, esize] = Elem[operand3, e, esize] + product;

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

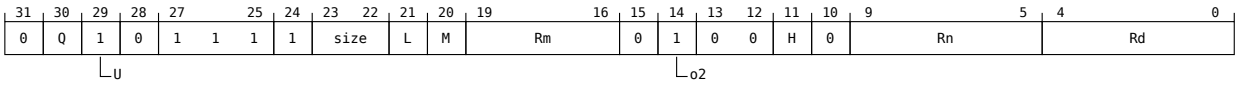
C7.2.182 MLS (by element)

Multiply-subtract from accumulator (vector, by element)

This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and subtracts the results from the vector elements of the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector
(FEAT_AdvSIMD)



Encoding

```
MLS <Vd>.<T>, <Vn>.<T>, V<m>.<Ts>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer idxdsize = 64 << UInt(H);
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L);   Rmhi = M;
    otherwise EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rmhi:Rm);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**m**> Is the number of the second SIMD&FP source register, encoded in "size:M:Rm":

size	<**m**>
00	RESERVED
01	UInt('0':Rm)
10	UInt(M:Rm)
11	RESERVED

Restricted to 0-15 when element size <Ts> is H.

<**Ts**> Is an element size specifier, encoded in "size":

size	<**Ts**>
00	RESERVED
01	H
10	S
11	RESERVED

<**index**> Is the element index, encoded in "size:H:L:M":

size	<**index**>
00	RESERVED
01	UInt(H:L:M)
10	UInt(H:L)
11	RESERVED

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(idxdsize) operand2 = V[m, idxdsize];
constant bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) product;

element2 = UInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    product = (element1 * element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[operand3, e, esize] - product;
V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

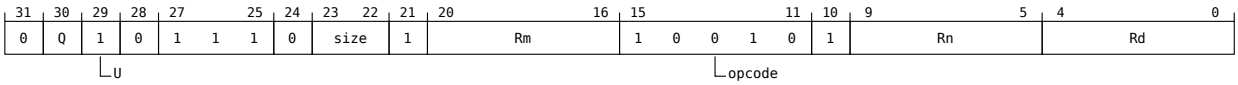
C7.2.183 MLS (vector)

Multiply-subtract from accumulator (vector)

This instruction multiplies corresponding elements in the vectors of the two source SIMD&FP registers, and subtracts the results from the vector elements of the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type
(FEAT_AdvSIMD)



Encoding

```
MLS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant bits(datasize) operand2 = V[m, datasize];
constant bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) product;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    product = (UInt(element1) * UInt(element2))<esize-1:0>;
    Elem[result, e, esize] = Elem[operand3, e, esize] - product;

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.184 MOV (scalar)

Move vector element to scalar

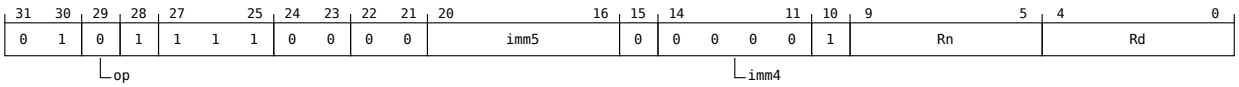
This instruction duplicates the specified vector element in the SIMD&FP source register into a scalar, and writes the result to the SIMD&FP destination register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of DUP (element). This means:

- The encodings in this description are named to match the encodings of DUP (element).
- The description of DUP (element) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Scalar
(FEAT_AdvSIMD)



Encoding

```
MOV <V><d>, <Vn>.<T>[<index>]
```

is equivalent to

```
DUP <V><d>, <Vn>.<T>[<index>]
```

and is always the preferred disassembly.

Assembler Symbols

<**V**> Is the destination width specifier, encoded in "imm5":

imm5	<V>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<**T**> Is the element width specifier, encoded in "imm5":

imm5	<T>
x0000	RESERVED

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

imm5	<T>
xxxx1	B
xxx10	H
xx100	S
x1000	D

<**index**> Is the element index encoded in “imm5”:

imm5	<index>
x0000	RESERVED
xxxx1	UInt(imm5<4:1>)
xxx10	UInt(imm5<4:2>)
xx100	UInt(imm5<4:3>)
x1000	UInt(imm5<4>)

Operation

The description of DUP (element) gives the operational pseudocode for this instruction.

Operational Information

The description of DUP (element) gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.185 MOV (element)

Move vector element to another vector element

This instruction copies the vector element of the source SIMD&FP register to the specified vector element of the destination SIMD&FP register.

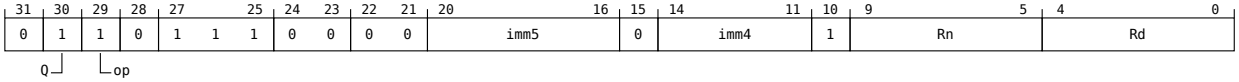
This instruction can insert data into individual elements within a SIMD&FP register without clearing the remaining bits to zero.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of INS (element). This means:

- The encodings in this description are named to match the encodings of INS (element).
- The description of INS (element) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Advanced SIMD
(FEAT_AdvSIMD)



Encoding

```
MOV <Vd>.<Ts>[<index1>], <Vn>.<Ts>[<index2>]
```

is equivalent to

```
INS <Vd>.<Ts>[<index1>], <Vn>.<Ts>[<index2>]
```

and is always the preferred disassembly.

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Ts**> Is an element size specifier, encoded in “imm5”:

imm5	<**Ts**>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

<**index1**> Is the destination element index encoded in “imm5”:

imm5	<**index1**>
x0000	RESERVED

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

imm5	<index1>
xxxx1	UInt(imm5<4:1>)
xxx10	UInt(imm5<4:2>)
xx100	UInt(imm5<4:3>)
x1000	UInt(imm5<4>)

<**Vn**> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<**index2**> Is the source element index encoded in “imm5:imm4”:

imm5	<index2>
x0000	RESERVED
xxxx1	UInt(imm4)
xxx10	UInt(imm4<3:1>)
xx100	UInt(imm4<3:2>)
x1000	UInt(imm4<3>)

Unspecified bits in “imm4” are ignored but should be set to zero by an assembler.

Operation

The description of INS (element) gives the operational pseudocode for this instruction.

Operational Information

The description of INS (element) gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.186 MOV (from general)

Move general-purpose register to a vector element

This instruction copies the contents of the source general-purpose register to the specified vector element in the destination SIMD&FP register.

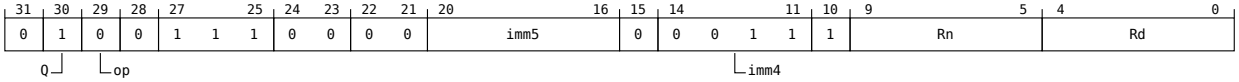
This instruction can insert data into individual elements within a SIMD&FP register without clearing the remaining bits to zero.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of INS (general). This means:

- The encodings in this description are named to match the encodings of INS (general).
- The description of INS (general) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Advanced SIMD
(FEAT_AdvSIMD)



Encoding

```
MOV <Vd>.<Ts>[<index>], <R><n>
```

is equivalent to

```
INS <Vd>.<Ts>[<index>], <R><n>
```

and is always the preferred disassembly.

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**Ts**> Is an element size specifier, encoded in "imm5":

imm5	<**Ts**>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

<**index**> Is the element index encoded in "imm5":

imm5	<**index**>
x0000	RESERVED

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

imm5	<index>
xxxx1	UInt(imm5<4:1>)
xxx10	UInt(imm5<4:2>)
xx100	UInt(imm5<4:3>)
x1000	UInt(imm5<4>)

<**R**> Is the width specifier for the general-purpose source register, encoded in “imm5”:

imm5	<R>
x0000	RESERVED
xxxx1	W
xxx10	W
xx100	W
x1000	X

<**n**> Is the number [0-30] of the general-purpose source register or ZR (31), encoded in the “Rn” field.

Operation

The description of INS (general) gives the operational pseudocode for this instruction.

Operational Information

The description of INS (general) gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.187 MOV (vector)

Move vector

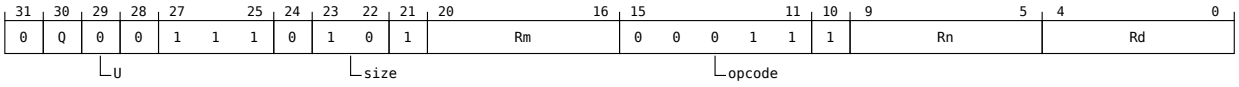
This instruction copies the vector in the source SIMD&FP register into the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of ORR (vector, register). This means:

- The encodings in this description are named to match the encodings of ORR (vector, register).
- The description of ORR (vector, register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Three registers of the same type
(FEAT_AdvSIMD)



Encoding

```
MOV <Vd>.<T>, <Vn>.<T>
```

is equivalent to

```
ORR <Vd>.<T>, <Vn>.<T>, <Vn>.<T>
```

and is the preferred disassembly when `Rm == Rn`.

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> Is an arrangement specifier, encoded in "Q":

Q	<T>
0	8B
1	16B

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

The description of ORR (vector, register) gives the operational pseudocode for this instruction.

Operational Information

The description of ORR (vector, register) gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.188 MOV (to general)

Move vector element to general-purpose register

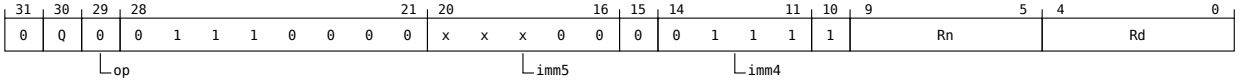
This instruction reads the unsigned integer from the source SIMD&FP register, zero-extends it to form a 32-bit or 64-bit value, and writes the result to the destination general-purpose register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of UMOV. This means:

- The encodings in this description are named to match the encodings of UMOV.
- The description of UMOV gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Advanced SIMD
(FEAT_AdvSIMD)



Encoding for the 32-bit variant Applies when `(Q == 0 && imm5 == xx100)`

```
MOV <Wd>, <Vn>.S[<index>]
```

is equivalent to

```
UMOV <Wd>, <Vn>.S[<index>]
```

and is always the preferred disassembly.

Encoding for the 64-bit variant Applies when `(Q == 1 && imm5 == x1000)`

```
MOV <Xd>, <Vn>.D[<index>]
```

is equivalent to

```
UMOV <Xd>, <Vn>.D[<index>]
```

and is always the preferred disassembly.

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Vn**> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<**index**> For the “32-bit” variant: is the element index encoded in “imm5<4:3>”.

For the “64-bit” variant: is the element index encoded in “imm5<4>”.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

Operation

The description of UMOV gives the operational pseudocode for this instruction.

Operational Information

The description of UMOV gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

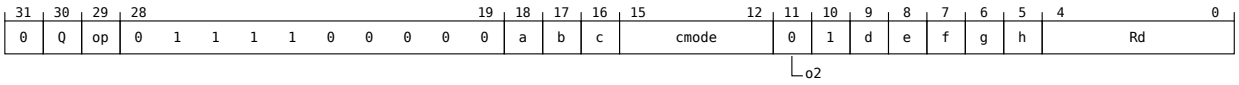
C7.2.189 MOVI

Move immediate (vector)

This instruction places an immediate constant into every vector element of the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Advanced SIMD
(FEAT_AdvSIMD)



Encoding for the 8-bit variant Applies when `(op == 0 && cmode == 1110)`

```
MOVI <Vd>.<T>, #<imm8>{, LSL #0}
```

Encoding for the 16-bit shifted immediate variant Applies when `(op == 0 && cmode == 10x0)`

```
MOVI <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

Encoding for the 32-bit shifted immediate variant Applies when `(op == 0 && cmode == 0xx0)`

```
MOVI <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

Encoding for the 32-bit shifting ones variant Applies when `(op == 0 && cmode == 110x)`

```
MOVI <Vd>.<T>, #<imm8>, MSL #<amount>
```

Encoding for the 64-bit scalar variant Applies when `(Q == 0 && op == 1 && cmode == 1110)`

```
MOVI <Dd>, #<imm>
```

Encoding for the 64-bit vector variant Applies when `(Q == 1 && op == 1 && cmode == 1110)`

```
MOVI <Vd>.2D, #<imm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer rd = UInt(Rd);
constant integer datasize = 64 << UInt(Q);
constant bits(64) imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
constant bits(datasize) imm = Replicate(imm64, datasize DIV 64);
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> For the "8-bit" variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	8B
1	16B

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For the “16-bit shifted immediate” variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the “32-bit shifted immediate” and “32-bit shifting ones” variants: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	2S
1	4S

<**imm8**> Is an 8-bit immediate encoded in “a:b:c:d:e:f:g:h”.

<**amount**> For the “16-bit shifted immediate” variant: is the shift amount encoded in “cmode<1>”:

cmode<1>	<amount>
0	0
1	8

defaulting to 0 if LSL is omitted.

For the “32-bit shifted immediate” variant: is the shift amount encoded in “cmode<2:1>”:

cmode<2:1>	<amount>
00	0
01	8
10	16
11	24

defaulting to 0 if LSL is omitted.

For the “32-bit shifting ones” variant: is the shift amount encoded in “cmode<0>”:

cmode<0>	<amount>
0	8
1	16

<**Dd**> Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**imm**> Is a 64-bit immediate ’aaaaaaaabbbbbbbbccccccccddddddddeeeeeeeeffffffffgggggggghhhhhhhh’, encoded in “a:b:c:d:e:f:g:h”.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
V[rd, datasize] = imm;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

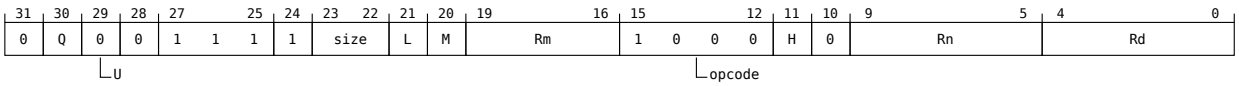
C7.2.190 MUL (by element)

Multiply (vector, by element)

This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector
(FEAT_AdvSIMD)



Encoding

```
MUL <Vd>.<T>, <Vn>.<T>, V<m>.<Ts>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer idxdsize = 64 << UInt(H);
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L);   Rmhi = M;
    otherwise EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rmhi:Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**m**> Is the number of the second SIMD&FP source register, encoded in "size:M:Rm":

size	<**m**>
00	RESERVED
01	UInt('0':Rm)
10	UInt(M:Rm)
11	RESERVED

Restricted to 0-15 when element size <Ts> is H.

<**Ts**> Is an element size specifier, encoded in "size":

size	<**Ts**>
00	RESERVED
01	H
10	S
11	RESERVED

<**index**> Is the element index, encoded in "size:H:L:M":

size	<**index**>
00	RESERVED
01	UInt(H:L:M)
10	UInt(H:L)
11	RESERVED

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(idxdsize) operand2 = V[m, idxdsize];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) product;

element2 = UInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    product = (element1 * element2)<esize-1:0>;
    Elem[result, e, esize] = product;

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

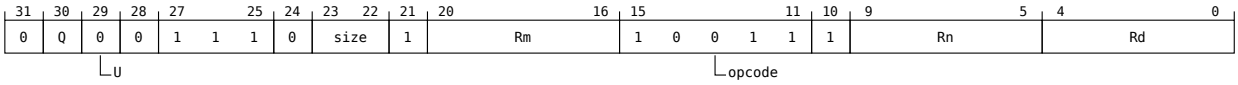
C7.2.191 MUL (vector)

Multiply (vector)

This instruction multiplies corresponding elements in the vectors of the two source SIMD&FP registers, places the results in a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type
(FEAT_AdvSIMD)



Encoding

```
MUL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if U == '1' && size != '00' then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) product;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    product = (UInt(element1) * UInt(element2))<esize-1:0>;
    Elem[result, e, esize] = product;

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.192 MVN

Bitwise NOT (vector)

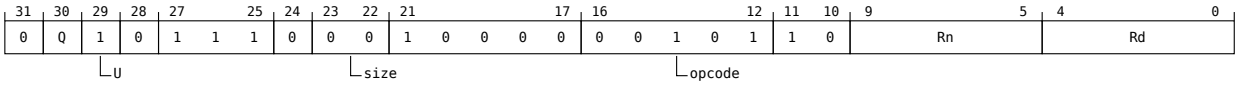
This instruction reads each vector element from the source SIMD&FP register, places the inverse of each value into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of NOT. This means:

- The encodings in this description are named to match the encodings of NOT.
- The description of NOT gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Vector
(FEAT_AdvSIMD)



Encoding

```
MVN <Vd>.<T>, <Vn>.<T>
```

is equivalent to

```
NOT <Vd>.<T>, <Vn>.<T>
```

and is always the preferred disassembly.

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	8B
1	16B

<**Vn**> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

The description of NOT gives the operational pseudocode for this instruction.

Operational Information

The description of NOT gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

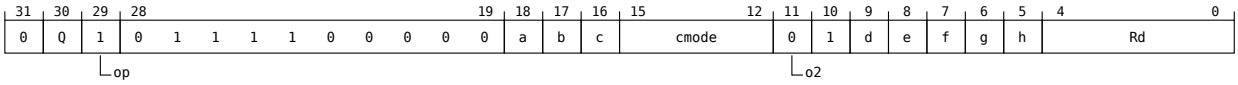
C7.2.193 MVNI

Move inverted immediate (vector)

This instruction places the inverse of an immediate constant into every vector element of the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Advanced SIMD
(FEAT_AdvSIMD)



Encoding for the 16-bit shifted immediate variant Applies when `(cmode == 10x0)`

```
MVNI <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

Encoding for the 32-bit shifted immediate variant Applies when `(cmode == 0xx0)`

```
MVNI <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

Encoding for the 32-bit shifting ones variant Applies when `(cmode == 110x)`

```
MVNI <Vd>.<T>, #<imm8>, MSL #<amount>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer rd = UInt(Rd);
constant integer datasize = 64 << UInt(Q);
constant bits(64) imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
constant bits(datasize) imm = Replicate(imm64, datasize DIV 64);
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> For the "16-bit shifted immediate" variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the "32-bit shifted immediate" and "32-bit shifting ones" variants: is an arrangement specifier, encoded in "Q":

Q	<T>
0	2S
1	4S

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**imm8**> Is an 8-bit immediate encoded in “a:b:c:d:e:f:g:h”.

<**amount**> For the “16-bit shifted immediate” variant: is the shift amount encoded in “cmode<1>”:

cmode<1>	<**amount**>
0	0
1	8

defaulting to 0 if LSL is omitted.

For the “32-bit shifted immediate” variant: is the shift amount encoded in “cmode<2:1>”:

cmode<2:1>	<**amount**>
00	0
01	8
10	16
11	24

defaulting to 0 if LSL is omitted.

For the “32-bit shifting ones” variant: is the shift amount encoded in “cmode<0>”:

cmode<0>	<**amount**>
0	8
1	16

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
V[rd, datasize] = NOT(imm);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.194 NEG (vector)

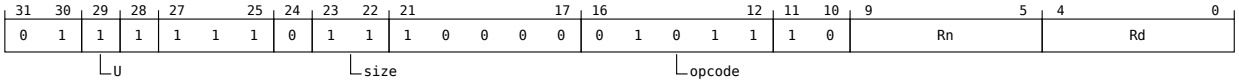
Negate (vector)

This instruction reads each vector element from the source SIMD&FP register, negates each value, puts the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



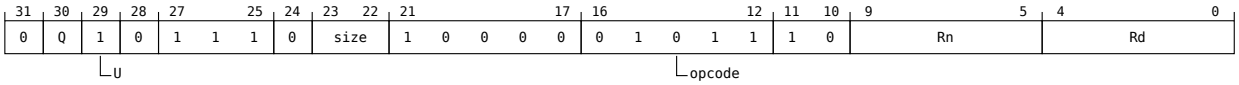
Encoding

```
NEG D<d>, D<n>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size != '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
NEG <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**n**> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    element = -element;
    Elem[result, e, esize] = element<esize-1:0>;

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.195 NOT

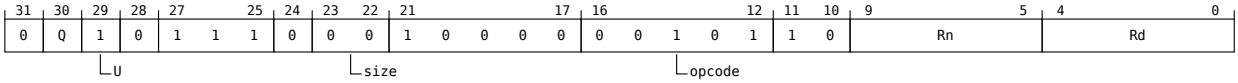
Bitwise NOT (vector)

This instruction reads each vector element from the source SIMD&FP register, places the inverse of each value into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias MVN.

Vector
(FEAT_AdvSIMD)



Encoding

```
NOT <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 8;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV 8;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "Q":

Q	<T>
0	8B
1	16B

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = NOT(element);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

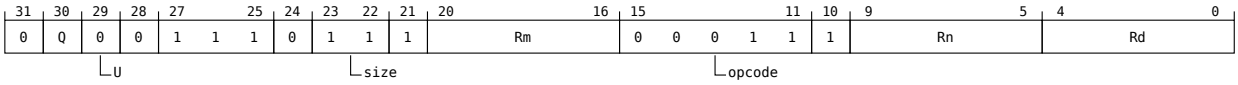
C7.2.196 ORN (vector)

Bitwise inclusive OR NOT (vector)

This instruction performs a bitwise OR NOT between the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type
(FEAT_AdvSIMD)



Encoding

```
ORN <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 64 << UInt(Q);
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	8B
1	16B

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = NOT(V[m, datasize]);
V[d, datasize] = operand1 OR operand2;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

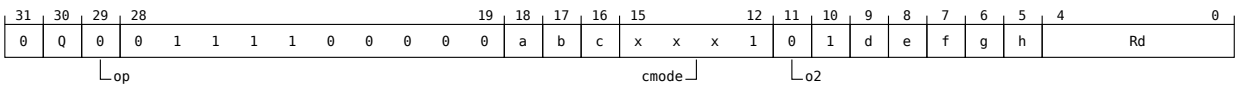
C7.2.197 ORR (vector, immediate)

Bitwise inclusive OR (vector, immediate)

This instruction reads each vector element from the destination SIMD&FP register, performs a bitwise OR between each result and an immediate constant, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Shifted immediate
(FEAT_AdvSIMD)



Encoding for the 16-bit variant Applies when `(cmode == 10x1)`

```
ORR <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

Encoding for the 32-bit variant Applies when `(cmode == 0xx1)`

```
ORR <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer rd = UInt(Rd);
constant integer datasize = 64 << UInt(Q);
constant bits(64) imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
constant bits(datasize) imm = Replicate(imm64, datasize DIV 64);
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP register, encoded in the "Rd" field.

<**T**> For the "16-bit" variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the "32-bit" variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	2S
1	4S

<**imm8**> Is an 8-bit immediate encoded in "a:b:c:d:e:f:g:h".

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**amount**> For the “16-bit” variant: is the shift amount encoded in “cmode<1>”:

cmode<1>	<**amount**>
0	0
1	8

defaulting to 0 if LSL is omitted.

For the “32-bit” variant: is the shift amount encoded in “cmode<2:1>”:

cmode<2:1>	<**amount**>
00	0
01	8
10	16
11	24

defaulting to 0 if LSL is omitted.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[rd, datasize];
V[rd, datasize] = operand OR imm;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.198 ORR (vector, register)

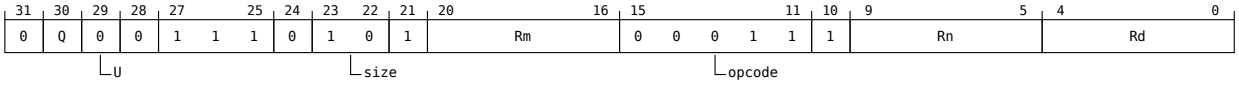
Bitwise inclusive OR (vector, register)

This instruction performs a bitwise OR between the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias MOV (vector).

Three registers of the same type
(FEAT_AdvSIMD)



Encoding

```
ORR <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 64 << UInt(Q);
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	8B
1	16B

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Alias Conditions

Alias	Is preferred when
MOV (vector)	Rm == Rn

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
V[d, datasize] = operand1 OR operand2;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.199 PMUL

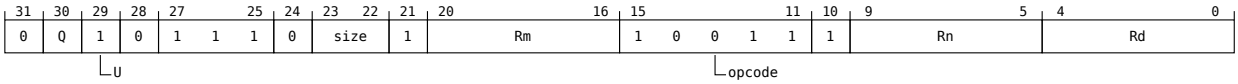
Polynomial multiply

This instruction multiplies corresponding elements in the vectors of the two source SIMD&FP registers, places the results in a vector, and writes the vector to the destination SIMD&FP register.

For information about multiplying polynomials see Polynomial arithmetic over {0, 1}.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type
(FEAT_AdvSIMD)



Encoding

```
PMUL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if U == '1' && size != '00' then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	x	RESERVED
1x	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
bits(esize) element1;
bits(esize) element2;
bits(esize) product;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    product = PolynomialMult(element1, element2)<esize-1:0>;
    Elem[result, e, esize] = product;

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.200 PMULL, PMULL2

Polynomial multiply long

This instruction multiplies corresponding elements in the lower or upper half of the vectors of the two source SIMD&FP registers, places the results in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

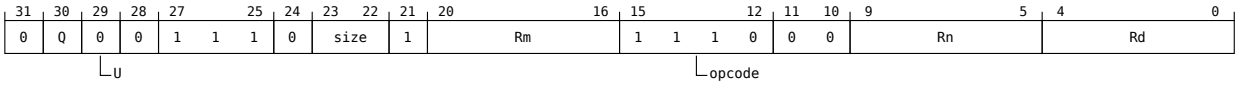
For information about multiplying polynomials, see Polynomial arithmetic over {0, 1}.

The PMULL instruction extracts each source vector from the lower half of each source register. The PMULL2 instruction extracts each source vector from the upper half of each source register.

The PMULL and PMULL2 variants that operate on 64-bit source elements are defined only when FEAT_PMULL is implemented.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers, not all the same type
(FEAT_AdvSIMD)



Encoding

```
PMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '01' || size == '10' then EndOfDecode(Decode_UNDEF);
if size == '11' && !IsFeatureImplemented(FEAT_PMULL) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Ta**> Is an arrangement specifier, encoded in “size”:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<Ta>
00	8H
01	RESERVED
10	RESERVED
11	1Q

The ’1Q’ arrangement is only allocated in an implementation that includes the Cryptographic Extension, and is otherwise RESERVED.

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	x	RESERVED
10	x	RESERVED
11	0	1D
11	1	2D

<Vm> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = Vpart[n, part, datasize];
constant bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, 2*esize] = PolynomialMult(element1, element2);

V[d, 2*datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.201 RADDHN, RADDHN2

Rounding add returning high narrow

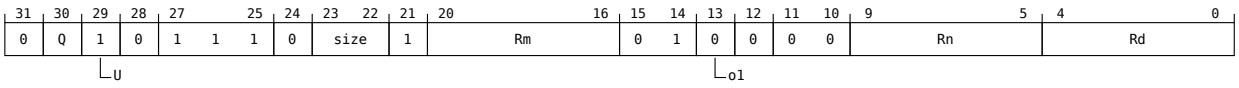
This instruction adds each vector element in the first source SIMD&FP register to the corresponding vector element in the second source SIMD&FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register.

The results are rounded. For truncated results, see ADDHN.

The `RADDHN` instruction writes the vector to the lower half of the destination register and clears the upper half. The `RADDHN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers, not all the same type
(FEAT_AdvSIMD)



Encoding

```
RADDHN{2} <Vd>.<Tb>, <Vn>.<Ta>, <Vm>.<Ta>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;
constant boolean round = TRUE;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Tb**> Is an arrangement specifier, encoded in “size:Q”:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Ta**> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(2*datasize) operand1 = V[n, 2*datasize];
constant bits(2*datasize) operand2 = V[m, 2*datasize];
bits(datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, 2*esize]);
    element2 = UInt(Elem[operand2, e, 2*esize]);
    sum = element1 + element2;
    sum = RShr(sum, esize, round);
    Elem[result, e, esize] = sum<esize-1:0>;

Vpart[d, part, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

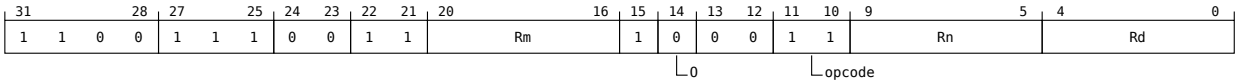
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.202 RAX1

Rotate and exclusive-OR

This instruction rotates each 64-bit element of the 128-bit vector in a source SIMD&FP register left by 1, performs a bitwise exclusive-OR of the resulting 128-bit vector and the vector in another source SIMD&FP register, and writes the result to the destination SIMD&FP register.

Advanced SIMD
(FEAT_SHA3)



Encoding

```
RAX1 <Vd>.2D, <Vn>.2D, <Vm>.2D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA3) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(128) Vm = V[m, 128];
constant bits(128) Vn = V[n, 128];
V[d, 128] = Vn EOR (ROL(Vm<127:64>, 1):ROL(Vm<63:0>, 1));
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

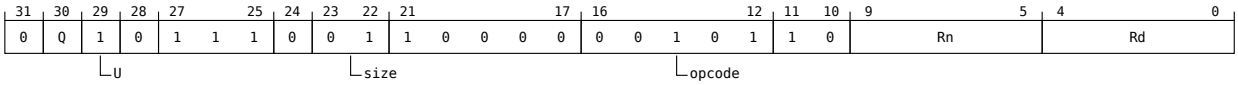
C7.2.203 RBIT (vector)

Reverse bit order (vector)

This instruction reads each vector element from the source SIMD&FP register, reverses the bits of the element, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector
(FEAT_AdvSIMD)



Encoding

```
RBIT <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 8;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV 8;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "Q":

Q	<T>
0	8B
1	16B

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;
bits(esize) rev;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    for i = 0 to esize-1
        rev<(esize-1)-i> = element<i>;
    Elem[result, e, esize] = rev;

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

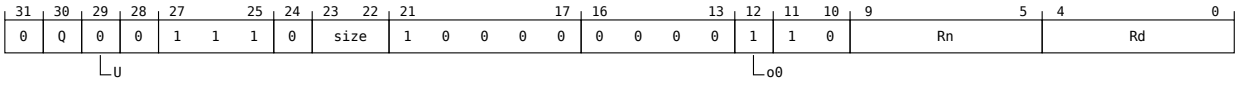
C7.2.204 REV16 (vector)

Reverse elements in 16-bit halfwords (vector)

This instruction reverses the order of 8-bit elements in each halfword of the vector in the source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector
(FEAT_AdvSIMD)



Encoding

```
REV16 <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer csize = 64 >> UInt(o0:U);
constant integer esize = 8 << UInt(size);
if csize <= esize then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer datasize = 64 << UInt(Q);
constant integer containers = datasize DIV csize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	x	RESERVED
1x	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(datasize) result;
for c = 0 to containers-1
    constant bits(csize) container = Elem[operand, c, csize];
    Elem[result, c, csize] = Reverse(container, esize);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

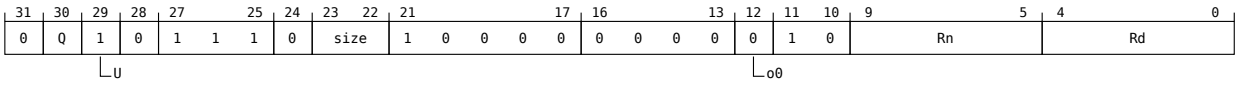
C7.2.205 REV32 (vector)

Reverse elements in 32-bit words (vector)

This instruction reverses the order of 8-bit or 16-bit elements in each word of the vector in the source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector
(FEAT_AdvSIMD)



Encoding

```
REV32 <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer csize = 64 >> UInt(o0:U);
constant integer esize = 8 << UInt(size);
if csize <= esize then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer datasize = 64 << UInt(Q);
constant integer containers = datasize DIV csize;
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
1x	x	RESERVED

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(datasize) result;
for c = 0 to containers-1
    constant bits(csize) container = Elem[operand, c, csize];
    Elem[result, c, csize] = Reverse(container, esize);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

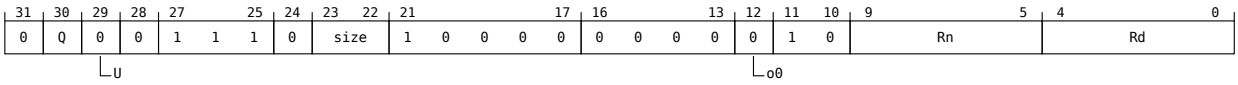
C7.2.206 REV64

Reverse elements in 64-bit doublewords (vector)

This instruction reverses the order of 8-bit, 16-bit, or 32-bit elements in each doubleword of the vector in the source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector
(FEAT_AdvSIMD)



Encoding

```
REV64 <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer csize = 64 >> UInt(o0:U);
constant integer esize = 8 << UInt(size);
if csize <= esize then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer datasize = 64 << UInt(Q);
constant integer containers = datasize DIV csize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(datasize) result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
for c = 0 to containers-1
    constant bits(csize) container = Elem[operand, c, csize];
    Elem[result, c, csize] = Reverse(container, esize);

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.207 RSHRN, RSHRN2

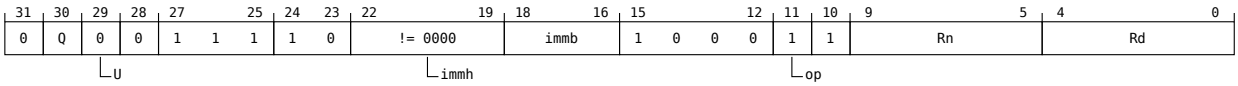
Rounding shift right narrow (immediate)

This instruction reads each unsigned integer value from the vector in the source SIMD&FP register, right shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements. The results are rounded. For truncated results, see SHRN.

The `RSHRN` instruction writes the vector to the lower half of the destination register and clears the upper half. The `RSHRN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector
(FEAT_AdvSIMD)



Encoding

```
RSHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh<3> == '1' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << HighestSetBitNZ(immh<2:0>);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;

constant integer shift = (2 * esize) - UInt(immh:immb);
constant boolean round = TRUE;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Tb**> Is an arrangement specifier, encoded in “immh:Q”:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

immh	Q	<Tb>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<**Ta**> Is an arrangement specifier, encoded in "immh":

immh	<Ta>
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

<**shift**> Is the right shift amount, in the range 1 to the destination element width in bits, encoded in "immh:immb":

immh	<shift>
0001	16 - UInt(immh:immb)
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	RESERVED

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize*2) operand = V[n, datasize*2];
bits(datasize) result;
integer element;

for e = 0 to elements-1
    element = RShr(UInt(Elem[operand, e, 2*esize]), shift, round);
    Elem[result, e, esize] = element<esize-1:0>;

Vpart[d, part, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.208 RSUBHN, RSUBHN2

Rounding subtract returning high narrow

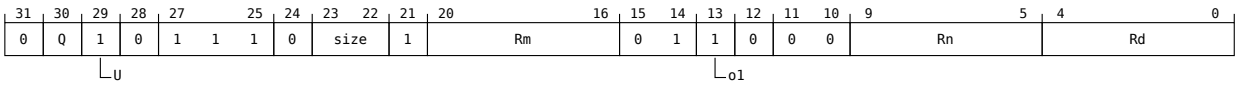
This instruction subtracts each vector element of the second source SIMD&FP register from the corresponding vector element of the first source SIMD&FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register.

The results are rounded. For truncated results, see SUBHN.

The `RSUBHN` instruction writes the vector to the lower half of the destination register and clears the upper half. The `RSUBHN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers, not all the same type
(FEAT_AdvSIMD)



Encoding

```
RSUBHN{2} <Vd>.<Tb>, <Vn>.<Ta>, <Vm>.<Ta>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;
constant boolean round = TRUE;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Tb**> Is an arrangement specifier, encoded in “size:Q”:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Ta**> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(2*datasize) operand1 = V[n, 2*datasize];
constant bits(2*datasize) operand2 = V[m, 2*datasize];
bits(datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, 2*esize]);
    element2 = UInt(Elem[operand2, e, 2*esize]);
    sum = element1 - element2;
    sum = RShr(sum, esize, round);
    Elem[result, e, esize] = sum<esize-1:0>;

Vpart[d, part, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

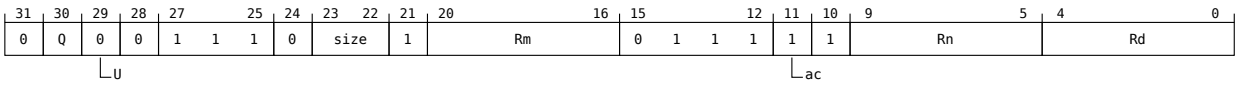
C7.2.209 SABA

Signed absolute difference and accumulate

This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, and accumulates the absolute values of the results into the elements of the vector of the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type
(FEAT_AdvSIMD)



Encoding

```
SABA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result = V[d, datasize];
integer element1;
integer element2;
bits(esize) absdiff;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    absdiff = Abs(element1 - element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.210 SABAL, SABAL2

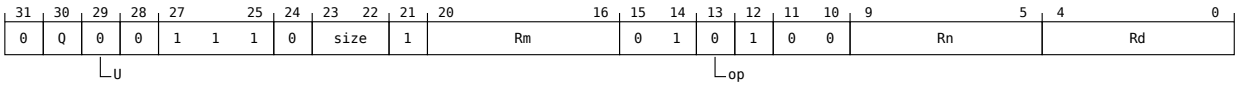
Signed absolute difference and accumulate long

This instruction subtracts the vector elements in the lower or upper half of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, and accumulates the absolute values of the results into the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The SABAL instruction extracts each source vector from the lower half of each source register. The SABAL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers, not all the same type
(FEAT_AdvSIMD)



Encoding

```
SABAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "size":

size	<Ta>
00	8H

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<Ta>
01	4S
10	2D
11	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Tb**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = Vpart[n, part, datasize];
constant bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result = V[d, 2*datasize];
integer element1;
integer element2;
bits(2*esize) absdiff;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    absdiff = Abs(element1-element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + absdiff;
V[d, 2*datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

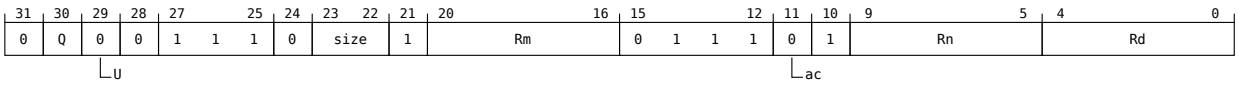
C7.2.211 SABD

Signed absolute difference

This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, places the absolute values of the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type
(FEAT_AdvSIMD)



Encoding

```
SABD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result = Zeros(datasize);
integer element1;
integer element2;
bits(esize) absdiff;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    absdiff = Abs(element1 - element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.212 SABDL, SABDL2

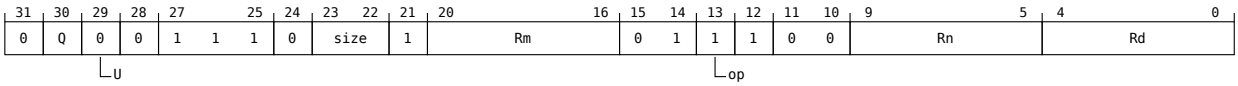
Signed absolute difference long

This instruction subtracts the vector elements in the lower or upper half of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, places the absolute value of the results into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The `SABDL` instruction extracts each source vector from the lower half of each source register. The `SABDL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers, not all the same type
(FEAT_AdvSIMD)



Encoding

```
SABDL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "size":

size	<Ta>
00	8H

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<Ta>
01	4S
10	2D
11	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Tb**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = Vpart[n, part, datasize];
constant bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result = Zeros(2*datasize);
integer element1;
integer element2;
bits(2*esize) absdiff;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    absdiff = Abs(element1-element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + absdiff;
V[d, 2*datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

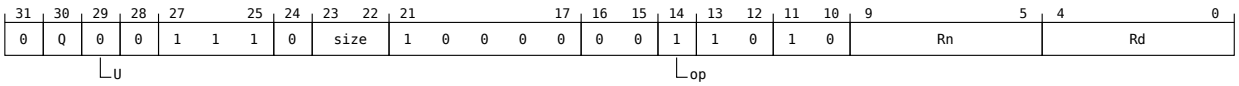
C7.2.213 SADALP

Signed add and accumulate long pairwise

This instruction adds pairs of adjacent signed integer values from the vector in the source SIMD&FP register and accumulates the results into the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector
(FEAT_AdvSIMD)



Encoding

```
SADALP <Vd>.<Ta>, <Vn>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV (2 * esize);
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Ta**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Ta>
00	0	4H
00	1	8H
01	0	2S
01	1	4S
10	0	1D
10	1	2D
11	x	RESERVED

<**Vn**> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<**Tb**> Is an arrangement specifier, encoded in “size:Q”:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(datasize) result  = V[d, datasize];

bits(2*esize) sum;
integer op1;
integer op2;

for e = 0 to elements-1
    op1 = SInt(Elem[operand, 2*e+0, esize]);
    op2 = SInt(Elem[operand, 2*e+1, esize]);
    sum = (op1+op2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.214 SADDL, SADDL2

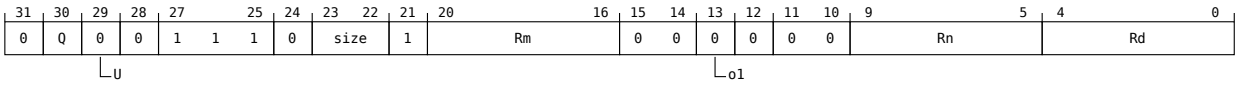
Signed add long (vector)

This instruction adds each vector element in the lower or upper half of the first source SIMD&FP register to the corresponding vector element of the second source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are signed integer values.

The SADDL instruction extracts each source vector from the lower half of each source register. The SADDL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers, not all the same type
(FEAT_AdvSIMD)



Encoding

```
SADDL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "size":

size	<Ta>
00	8H

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<Ta>
01	4S
10	2D
11	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Tb**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = Vpart[n, part, datasize];
constant bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d, 2*datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

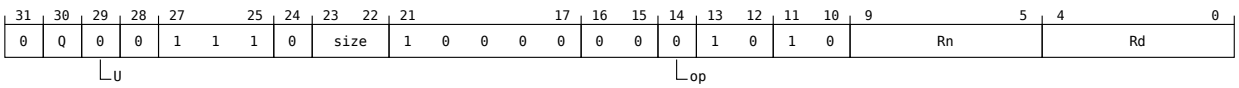
C7.2.215 SADDLP

Signed add long pairwise

This instruction adds pairs of adjacent signed integer values from the vector in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector
(FEAT_AdvSIMD)



Encoding

```
SADDLP <Vd>.<Ta>, <Vn>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV (2 * esize);
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Ta**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Ta>
00	0	4H
00	1	8H
01	0	2S
01	1	4S
10	0	1D
10	1	2D
11	x	RESERVED

<**Vn**> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<**Tb**> Is an arrangement specifier, encoded in “size:Q”:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(datasize) result;

bits(2*esize) sum;
integer op1;
integer op2;

for e = 0 to elements-1
    op1 = SInt(Elem[operand, 2*e+0, esize]);
    op2 = SInt(Elem[operand, 2*e+1, esize]);
    sum = (op1+op2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = sum;

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

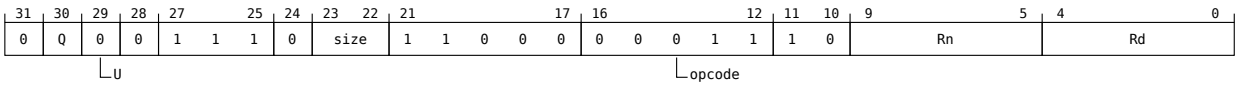
C7.2.216 SADDLV

Signed add long across vector

This instruction adds every vector element in the source SIMD&FP register together, and writes the scalar result to the destination SIMD&FP register. The destination scalar is twice as long as the source vector elements. All the values in this instruction are signed integer values.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Advanced SIMD
(FEAT_AdvSIMD)



Encoding

```
SADDLV <V><d>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '100' then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;

constant boolean unsigned = FALSE;
```

Assembler Symbols

<**V**> Is the destination width specifier, encoded in “size”:

size	<**V**>
00	H
01	S
10	D
11	RESERVED

<**d**> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<**Vn**> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<**T**> Is an arrangement specifier, encoded in “size:Q”:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	RESERVED
10	1	4S
11	x	RESERVED

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(esize) opelt = Elem[operand, 0, esize];
integer sum = if unsigned then UInt(opelt) else SInt(opelt);

for e = 1 to elements-1
    opelt = Elem[operand, e, esize];
    constant integer element = if unsigned then UInt(opelt) else SInt(opelt);
    sum = sum + element;

V[d, 2*esize] = sum<2*esize-1:0>;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.217 SADDW, SADDW2

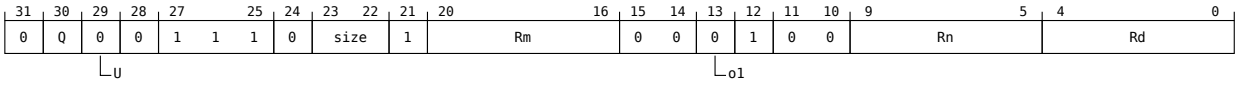
Signed add wide

This instruction adds vector elements of the first source SIMD&FP register to the corresponding vector elements in the lower or upper half of the second source SIMD&FP register, places the results in a vector, and writes the vector to the SIMD&FP destination register.

The `SADDW` instruction extracts the second source vector from the lower half of the second source register. The `SADDW2` instruction extracts the second source vector from the upper half of the second source register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers, not all the same type
(FEAT_AdvSIMD)



Encoding

```
SADDW{2} <Vd>.<Ta>, <Vn>.<Ta>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "size":

size	<Ta>
00	8H
01	4S

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<Ta>
10	2D
11	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

<**Tb**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(2*datasize) operand1 = V[n, 2*datasize];
constant bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, 2*esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d, 2*datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.218 SCVTF (vector, fixed-point)

Signed fixed-point convert to floating-point (vector)

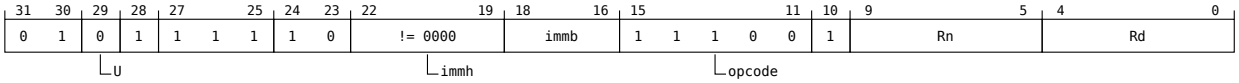
This instruction converts each element in a vector from fixed-point to floating-point using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



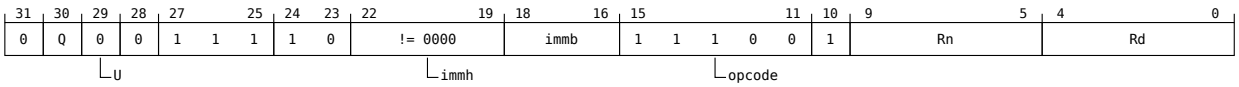
Encoding

```
SCVTF <V><d>, <V><n>, #<fbits>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh IN {'000x'} || (immh IN {'001x'} && !IsFeatureImplemented(FEAT_FP16)) then
    EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = if immh IN {'1xxx'} then 64 else if immh IN {'01xx'} then 32 else 16;
constant integer datasize = esize;
constant integer elements = 1;

constant integer fracbits = (esize * 2) - UInt(immh:immb);
constant boolean unsigned = FALSE;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
SCVTF <Vd>.<T>, <Vn>.<T>, #<fbits>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh IN {'000x'} || (immh IN {'001x'} && !IsFeatureImplemented(FEAT_FP16)) then
    EndOfDecode(Decode_UNDEF);
if immh<3>:Q == '10' then EndOfDecode(Decode_UNDEF);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = if immh IN {'1xxx'} then 64 else if immh IN {'01xx'} then 32 else 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;

constant integer fracbits = (esize * 2) - UInt(immh:immb);
constant boolean unsigned = FALSE;
```

Assembler Symbols

<**V**> Is a width specifier, encoded in "immh":

immh	<V>	Architectural Feature
0001	RESERVED	-
001x	H	FEAT_FP16
01xx	S	-
1xxx	D	-

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**n**> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<**fbits**> For the "Scalar" variant: is the number of fractional bits, in the range 1 to the operand width, encoded in "immh:immb":

immh	<fbits>
0001	RESERVED
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	128 - UInt(immh:immb)

For the "Vector" variant: is the number of fractional bits, in the range 1 to the element width, encoded in "immh:immb":

immh	<fbits>
0001	RESERVED
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	128 - UInt(immh:immb)

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> Is an arrangement specifier, encoded in "immh:Q":

immh	Q	<T>	Architectural Feature
0001	x	RESERVED	-
001x	0	4H	FEAT_FP16

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

immh	Q	<T>	Architectural Feature
001x	1	8H	FEAT_FP16
01xx	0	2S	-
01xx	1	4S	-
1xxx	0	RESERVED	-
1xxx	1	2D	-

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand  = V[n, datasize];
constant FPRounding rounding = FPRoundingMode(FPCR);

bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FixedToFP(element, fracbits, unsigned, FPCR, rounding, esize);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.219 SCVTF (vector, integer)

Signed integer convert to floating-point (vector)

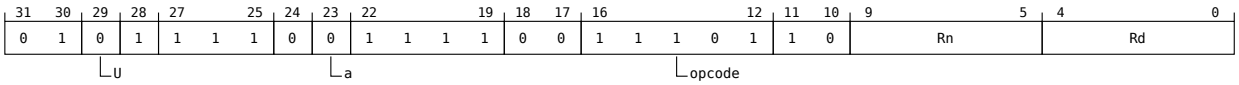
This instruction converts each element in a vector from signed integer to floating-point using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision , Scalar single-precision and double-precision , Vector half-precision and Vector single-precision and double-precision

Scalar half-precision
(FEAT_AdvSIMD && FEAT_FP16)



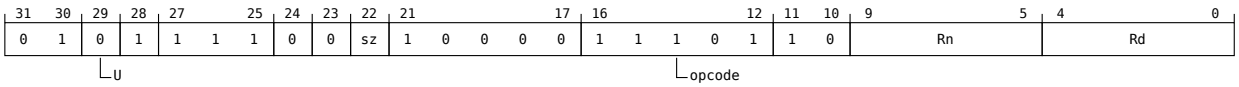
Encoding

```
SCVTF <Hd>, <Hn>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 16;
constant integer datasize = esize;
constant integer elements = 1;

constant boolean unsigned = FALSE;
```

Scalar single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

```
SCVTF <V><d>, <V><n>
```

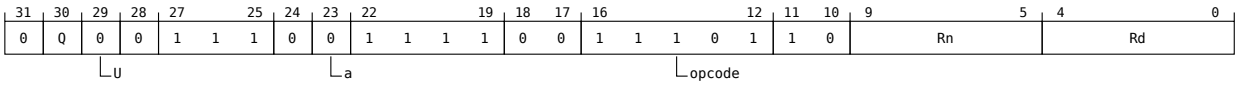
Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 32 << UInt(sz);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer datasize = esize;
constant integer elements = 1;
constant boolean unsigned = FALSE;
```

Vector half-precision (FEAT_AdvSIMD && FEAT_FP16)



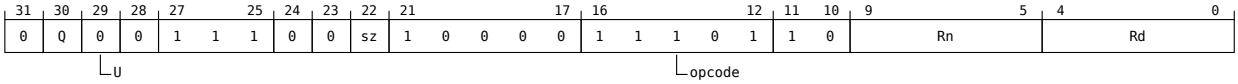
Encoding

```
SCVTF <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;

constant boolean unsigned = FALSE;
```

Vector single-precision and double-precision (FEAT_AdvSIMD)



Encoding

```
SCVTF <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;

constant boolean unsigned = FALSE;
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**V**> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**n**> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> For the "Vector half-precision" variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the "Vector single-precision and double-precision" variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(datasize) operand = V[n, datasize];

bits(datasize) result;
constant FPRounding rounding = FPRoundingMode(FPCR);
constant integer fracbits = 0;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FixedToFP(element, fracbits, unsigned, FPCR, rounding, esize);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

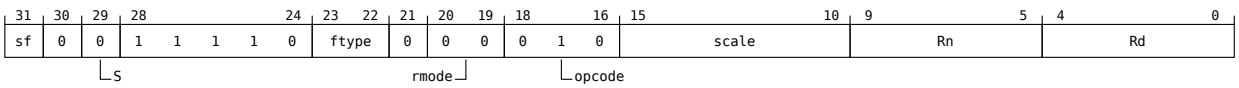
C7.2.220 SCVTF (scalar, fixed-point)

Signed fixed-point convert to floating-point (scalar)

This instruction converts the signed value in the 32-bit or 64-bit general-purpose source register to a floating-point value using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Encoding for the 32-bit to half-precision variant
(FEAT_FP16) Applies when `(sf == 0 && ftype == 11)`

```
SCVTF <Hd>, <Wn>, #<fbits>
```

Encoding for the 64-bit to half-precision variant
(FEAT_FP16) Applies when `(sf == 1 && ftype == 11)`

```
SCVTF <Hd>, <Xn>, #<fbits>
```

Encoding for the 32-bit to single-precision variant
(FEAT_FP) Applies when `(sf == 0 && ftype == 00)`

```
SCVTF <Sd>, <Wn>, #<fbits>
```

Encoding for the 64-bit to single-precision variant
(FEAT_FP) Applies when `(sf == 1 && ftype == 00)`

```
SCVTF <Sd>, <Xn>, #<fbits>
```

Encoding for the 32-bit to double-precision variant
(FEAT_FP) Applies when `(sf == 0 && ftype == 01)`

```
SCVTF <Dd>, <Wn>, #<fbits>
```

Encoding for the 64-bit to double-precision variant
(FEAT_FP) Applies when `(sf == 1 && ftype == 01)`

```
SCVTF <Dd>, <Xn>, #<fbits>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if ftype == '10' then EndOfDecode(Decode_UNDEF);
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);
if sf == '0' && scale<5> == '0' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer intsize = 32 << UInt(sf);
constant integer decode_fltsize = 8 << UInt(ftype EOR '10');
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer fracbits = 64 - UInt(scale);

constant boolean unsigned = FALSE;
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<**Wn**> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

<**fbits**> For the "32-bit to double-precision", "32-bit to half-precision", and "32-bit to single-precision" variants: is the number of bits after the binary point in the fixed-point source, in the range 1 to 32, encoded as 64 minus "scale".

For the "64-bit to double-precision", "64-bit to half-precision", and "64-bit to single-precision" variants: is the number of bits after the binary point in the fixed-point source, in the range 1 to 64, encoded as 64 minus "scale".

<**Xn**> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

<**Sd**> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<**Dd**> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant integer fltsize = decode_fltsize;
constant FPRounding rounding = FPRoundingMode(FPCR);

constant bits(intsize) intval = X[n, intsize];
constant bits(fltsize) fltval = FixedToFP(intval, fracbits, unsigned, FPCR, rounding, fltsize);

V[d, fltsize] = fltval;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

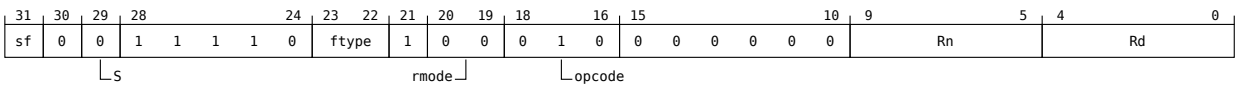
C7.2.221 SCVTF (scalar, integer)

Signed integer convert to floating-point (scalar)

This instruction converts the signed integer value in the general-purpose source register to a floating-point value using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Encoding for the 32-bit to half-precision variant
(FEAT_FP16) Applies when `(sf == 0 && ftype == 11)`

```
SCVTF <Hd>, <Wn>
```

Encoding for the 32-bit to single-precision variant
(FEAT_FP) Applies when `(sf == 0 && ftype == 00)`

```
SCVTF <Sd>, <Wn>
```

Encoding for the 32-bit to double-precision variant
(FEAT_FP) Applies when `(sf == 0 && ftype == 01)`

```
SCVTF <Dd>, <Wn>
```

Encoding for the 64-bit to half-precision variant
(FEAT_FP16) Applies when `(sf == 1 && ftype == 11)`

```
SCVTF <Hd>, <Xn>
```

Encoding for the 64-bit to single-precision variant
(FEAT_FP) Applies when `(sf == 1 && ftype == 00)`

```
SCVTF <Sd>, <Xn>
```

Encoding for the 64-bit to double-precision variant
(FEAT_FP) Applies when `(sf == 1 && ftype == 01)`

```
SCVTF <Dd>, <Xn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if ftype == '10' then EndOfDecode(Decode_UNDEF);
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer intsize = 32 << UInt(sf);
constant integer decode_fltsize = 8 << UInt(ftype EOR '10');
constant boolean unsigned    = FALSE;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Wn**> Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<**Sd**> Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Dd**> Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Xn**> Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant integer fltsize = decode_fltsize;

constant bits(intsize) intval = X[n, intsize];
constant FPRounding rounding  = FPRoundingMode(FPCR);
constant integer fracbits = 0;
constant bits(fltsize) fltval = FixedToFP(intval, fracbits, unsigned, FPCR, rounding, fltsize);

V[d, fltsize] = fltval;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

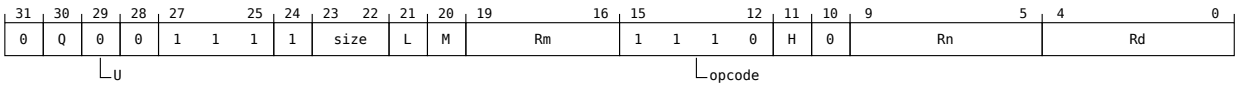
C7.2.222 SDOT (by element)

Dot product signed arithmetic (vector, by element)

This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of an indexed 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector
(FEAT_DotProd)



Encoding

```
SDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.4B[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_DotProd) then EndOfDecode(Decode_UNDEF);
if size  != '10' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(M:Rm);
constant integer index = UInt(H:L);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP third source and destination register, encoded in the “Rd” field.

<**Ta**> Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	2S
1	4S

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Tb**> Is an arrangement specifier, encoded in “Q”:

Q	<Tb>
0	8B
1	16B

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “M:Rm” fields.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**index**> Is the element index, encoded in the “H:L” fields.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(128) operand2 = V[m, 128];
bits(datasize) result = V[d, datasize];
for e = 0 to elements-1
    integer res = 0;
    integer element1, element2;
    for i = 0 to 3
        element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
        element2 = SInt(Elem[operand2, 4 * index + i, esize DIV 4]);

        res = res + element1 * element2;
    Elem[result, e, esize] = Elem[result, e, esize] + res;
V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

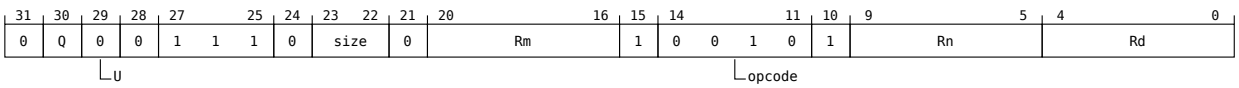
C7.2.223 SDOT (vector)

Dot product signed arithmetic (vector)

This instruction performs the dot product of the four signed 8-bit elements in each 32-bit element of the first source register with the four signed 8-bit elements of the corresponding 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector
(FEAT_DotProd)



Encoding

```
SDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_DotProd) then EndOfDecode(Decode_UNDEF);
if size != '10' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP third source and destination register, encoded in the “Rd” field.

<**Ta**> Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	2S
1	4S

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Tb**> Is an arrangement specifier, encoded in “Q”:

Q	<Tb>
0	8B
1	16B

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

result = V[d, datasize];
for e = 0 to elements-1
    integer res = 0;
    integer element1, element2;
    for i = 0 to 3
        element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
        element2 = SInt(Elem[operand2, 4 * e + i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = Elem[result, e, esize] + res;
V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

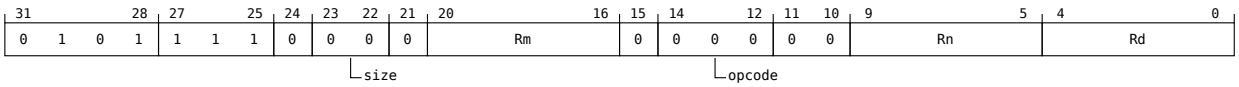
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.224 SHA1C

SHA1 hash update (choose)

SHA1 hash update (choose).

Advanced SIMD
(FEAT_SHA1)



Encoding

```
SHA1C <Qd>, <Sn>, <Vm>.4S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA1) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
```

Assembler Symbols

<Qd> Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) x = V[d, 128];
bits(32)  y = V[n, 32];     // Note: 32 not 128 bits wide
constant bits(128) w = V[m, 128];

for e = 0 to 3
    constant bits(32) t = SHAchoose(x<63:32>, x<95:64>, x<127:96>);
    y = y + ROL(x<31:0>, 5) + t + Elem[w, e, 32];
    x<63:32> = ROL(x<63:32>, 30);
    constant bits(160) yx = ROL(y:x, 32);
    (y, x) = (yx<159:128>, yx<127:0>);
V[d, 128] = x;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

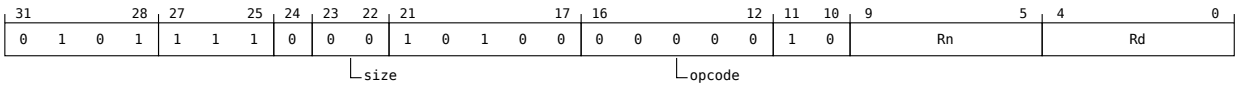
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.225 SHA1H

SHA1 fixed rotate

SHA1 fixed rotate.

Advanced SIMD
(FEAT_SHA1)



Encoding

```
SHA1H <Sd>, <Sn>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA1) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
```

Assembler Symbols

<**Sd**> Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Sn**> Is the 32-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(32) operand = V[n, 32];        // read element [0] only,  [1-3] zeroed
V[d, 32] = ROL(operand, 30);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

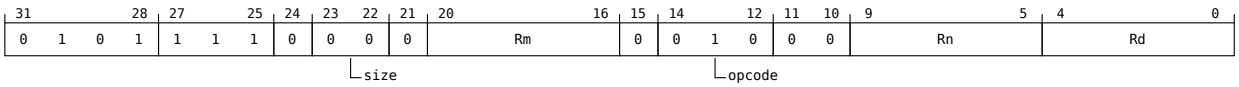
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.226 SHA1M

SHA1 hash update (majority)

SHA1 hash update (majority).

Advanced SIMD
(FEAT_SHA1)



Encoding

```
SHA1M <Qd>, <Sn>, <Vm>.4S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA1) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
```

Assembler Symbols

<Qd> Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) x = V[d, 128];
bits(32)  y = V[n, 32];     // Note: 32 not 128 bits wide
constant bits(128) w = V[m, 128];

for e = 0 to 3
    constant bits(32) t = SHAmajority(x<63:32>, x<95:64>, x<127:96>);
    y = y + ROL(x<31:0>, 5) + t + Elem[w, e, 32];
    x<63:32> = ROL(x<63:32>, 30);
    constant bits(160) yx = ROL(y:x, 32);
    (y, x) = (yx<159:128>, yx<127:0>);
V[d, 128] = x;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

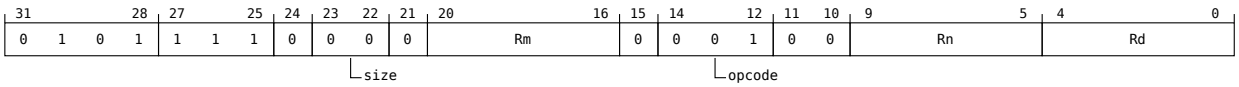
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.227 SHA1P

SHA1 hash update (parity)

SHA1 hash update (parity).

Advanced SIMD
(FEAT_SHA1)



Encoding

```
SHA1P <Qd>, <Sn>, <Vm>.4S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA1) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
```

Assembler Symbols

<Qd> Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) x = V[d, 128];
bits(32)  y = V[n, 32];     // Note: 32 not 128 bits wide
constant bits(128) w = V[m, 128];

for e = 0 to 3
    constant bits(32) t = SHAparity(x<63:32>, x<95:64>, x<127:96>);
    y = y + ROL(x<31:0>, 5) + t + Elem[w, e, 32];
    x<63:32> = ROL(x<63:32>, 30);
    constant bits(160) yx = ROL(y:x, 32);
    (y, x) = (yx<159:128>, yx<127:0>);
V[d, 128] = x;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

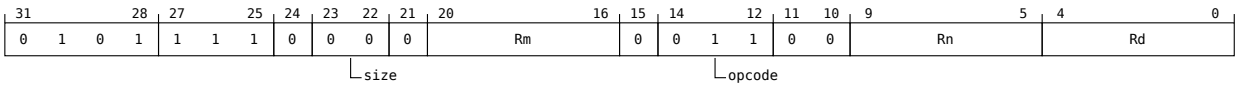
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.228 SHA1SU0

SHA1 schedule update 0

SHA1 schedule update 0.

Advanced SIMD
(FEAT_SHA1)



Encoding

```
SHA1SU0 <Vd>.4S, <Vn>.4S, <Vm>.4S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA1) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.

<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(128) operand1 = V[d, 128];
constant bits(128) operand2 = V[n, 128];
constant bits(128) operand3 = V[m, 128];
bits(128) result = operand2<63:0> : operand1<127:64>;
result = result EOR operand1 EOR operand3;
V[d, 128] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

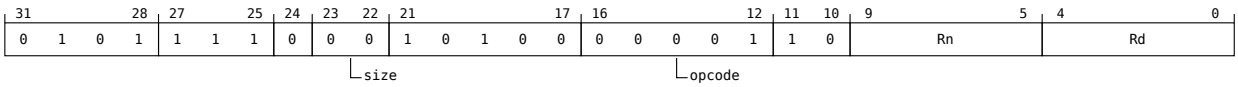
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.229 SHA1SU1

SHA1 schedule update 1

SHA1 schedule update 1.

Advanced SIMD
(FEAT_SHA1)



Encoding

```
SHA1SU1 <Vd>.4S, <Vn>.4S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA1) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.

<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(128) operand1 = V[d, 128];
constant bits(128) operand2 = V[n, 128];
constant bits(128) T = operand1 EOR LSR(operand2, 32);

bits(128) result;
result<31:0>   = ROL(T<31:0>,   1);
result<63:32>  = ROL(T<63:32>,  1);
result<95:64>  = ROL(T<95:64>,  1);
result<127:96> = ROL(T<127:96>, 1) EOR ROL(T<31:0>, 2);
V[d, 128] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

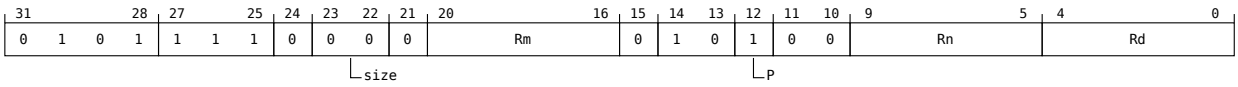
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.230 SHA256H2

SHA256 hash update (part 2)

SHA256 hash update (part 2).

Advanced SIMD
(FEAT_SHA256)



Encoding

```
SHA256H2 <Qd>, <Qn>, <Vm>.4S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA256) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
```

Assembler Symbols

<Qd> Is the 128-bit name of the SIMD&FP source and destination, encoded in the “Rd” field.

<Qn> Is the 128-bit name of the second SIMD&FP source register, encoded in the “Rn” field.

<Vm> Is the name of the third SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant boolean part1 = FALSE;
V[d, 128] = SHA256hash(V[n, 128], V[d, 128], V[m, 128], part1);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

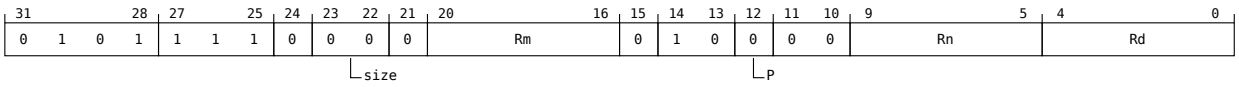
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.231 SHA256H

SHA256 hash update (part 1)

SHA256 hash update (part 1).

Advanced SIMD
(FEAT_SHA256)



Encoding

```
SHA256H <Qd>, <Qn>, <Vm>.4S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA256) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
```

Assembler Symbols

<Qd> Is the 128-bit name of the SIMD&FP source and destination, encoded in the “Rd” field.

<Qn> Is the 128-bit name of the second SIMD&FP source register, encoded in the “Rn” field.

<Vm> Is the name of the third SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant boolean part1 = TRUE;
V[d, 128] = SHA256hash(V[d, 128], V[n, 128], V[m, 128], part1);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

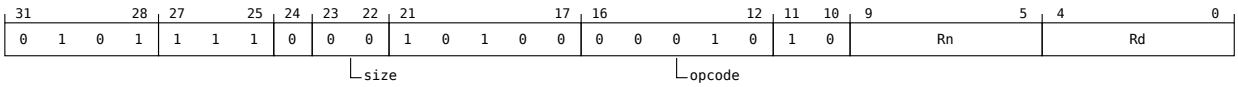
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.232 SHA256SU0

SHA256 schedule update 0

SHA256 schedule update 0.

Advanced SIMD
(FEAT_SHA256)



Encoding

```
SHA256SU0 <Vd>.4S, <Vn>.4S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA256) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.

<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(128) operand1 = V[d, 128];
constant bits(128) operand2 = V[n, 128];
constant bits(128) T = operand2<31:0> : operand1<127:32>;
bits(128) result;
bits(32) elt;

for e = 0 to 3
    elt = Elem[T, e, 32];
    elt = ROR(elt, 7) EOR ROR(elt, 18) EOR LSR(elt, 3);
    Elem[result, e, 32] = elt + Elem[operand1, e, 32];
V[d, 128] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

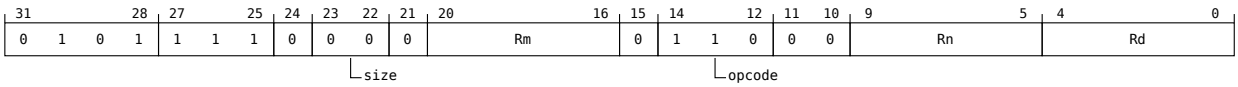
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.233 SHA256SU1

SHA256 schedule update 1

SHA256 schedule update 1.

Advanced SIMD
(FEAT_SHA256)



Encoding

```
SHA256SU1 <Vd>.4S, <Vn>.4S, <Vm>.4S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA256) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.

<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(128) operand1 = V[d, 128];
constant bits(128) operand2 = V[n, 128];
constant bits(128) operand3 = V[m, 128];
constant bits(128) T0 = operand3<31:0> : operand2<127:32>;

bits(64) T1;
bits(32) elt;
bits(128) result;

T1 = operand3<127:64>;
for e = 0 to 1
    elt = Elem[T1, e, 32];
    elt = ROR(elt, 17) EOR ROR(elt, 19) EOR LSR(elt, 10);
    elt = elt + Elem[operand1, e, 32] + Elem[T0, e, 32];
    Elem[result, e, 32] = elt;

T1 = result<63:0>;
for e = 2 to 3
    elt = Elem[T1, e - 2, 32];
    elt = ROR(elt, 17) EOR ROR(elt, 19) EOR LSR(elt, 10);
    elt = elt + Elem[operand1, e, 32] + Elem[T0, e, 32];
    Elem[result, e, 32] = elt;

V[d, 128] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

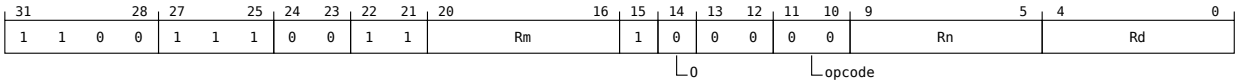
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.234 SHA512H

SHA512 hash update part 1

This instruction takes the values from the three 128-bit source SIMD&FP registers and produces a 128-bit output value that combines the sigma1 and chi functions of two iterations of the SHA512 calculation. It returns this value to the destination SIMD&FP register.

Advanced SIMD
(FEAT_SHA512)



Encoding

```
SHA512H <Qd>, <Qn>, <Vm>.2D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA512) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
```

Assembler Symbols

<Qd> Is the 128-bit name of the SIMD&FP source and destination register, encoded in the "Rd" field.

<Qn> Is the 128-bit name of the second SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vtmp;
bits(64)  MSigma1;
bits(64)  tmp;
constant bits(128) x = V[n, 128];
constant bits(128) y = V[m, 128];
constant bits(128) w = V[d, 128];

MSigma1 =  ROR(y<127:64>, 14) EOR ROR(y<127:64>, 18) EOR ROR(y<127:64>, 41);
Vtmp<127:64> =  (y<127:64> AND x<63:0>) EOR (NOT(y<127:64>) AND x<127:64>);
Vtmp<127:64> = (Vtmp<127:64> + MSigma1 +  w<127:64>);
tmp = Vtmp<127:64> + y<63:0>;
MSigma1 = ROR(tmp, 14) EOR ROR(tmp, 18) EOR ROR(tmp, 41);
Vtmp<63:0> = (tmp AND y<127:64>) EOR (NOT(tmp) AND x<63:0>);
Vtmp<63:0> = (Vtmp<63:0> + MSigma1 + w<63:0>);
V[d, 128] =  Vtmp;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

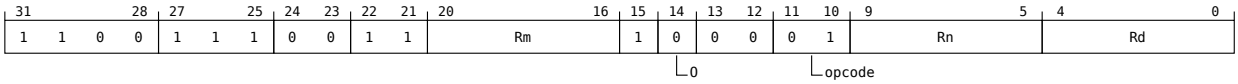
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.235 SHA512H2

SHA512 hash update part 2

This instruction takes the values from the three 128-bit source SIMD&FP registers and produces a 128-bit output value that combines the sigma0 and majority functions of two iterations of the SHA512 calculation. It returns this value to the destination SIMD&FP register.

Advanced SIMD
(FEAT_SHA512)



Encoding

```
SHA512H2 <Qd>, <Qn>, <Vm>.2D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA512) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
```

Assembler Symbols

<Qd> Is the 128-bit name of the SIMD&FP source and destination register, encoded in the "Rd" field.

<Qn> Is the 128-bit name of the second SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vtmp;
bits(64) NSigma0;
constant bits(128) x = V[n, 128];
constant bits(128) y = V[m, 128];
constant bits(128) w = V[d, 128];

NSigma0 =  ROR(y<63:0>, 28) EOR ROR(y<63:0>, 34) EOR ROR(y<63:0>, 39);
Vtmp<127:64> = SHAmajority(x<63:0>, y<127:64>, y<63:0>);
Vtmp<127:64> = (Vtmp<127:64> + NSigma0 +  w<127:64>);
NSigma0 =  ROR(Vtmp<127:64>, 28) EOR ROR(Vtmp<127:64>, 34) EOR ROR(Vtmp<127:64>, 39);
Vtmp<63:0> = SHAmajority(Vtmp<127:64>, y<63:0>, y<127:64>);
Vtmp<63:0> =   (Vtmp<63:0> + NSigma0 + w<63:0>);

V[d, 128] = Vtmp;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

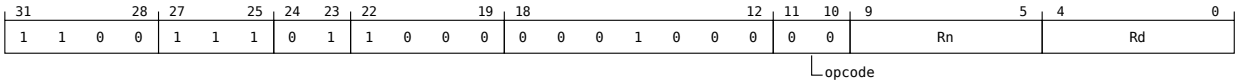
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.236 SHA512SU0

SHA512 schedule update 0

This instruction takes the values from the two 128-bit source SIMD&FP registers and produces a 128-bit output value that combines the gamma0 functions of two iterations of the SHA512 schedule update that are performed after the first 16 iterations within a block. It returns this value to the destination SIMD&FP register.

Advanced SIMD
(FEAT_SHA512)



Encoding

```
SHA512SU0 <Vd>.2D, <Vn>.2D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA512) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.

<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(64) sig0;
bits(128) Vtmp;
constant bits(128) x = V[n, 128];
constant bits(128) w = V[d, 128];
sig0 = ROR(w<127:64>, 1) EOR ROR(w<127:64>, 8) EOR ('0000000':w<127:71>);
Vtmp<63:0> = w<63:0> + sig0;
sig0 = ROR(x<63:0>, 1) EOR ROR(x<63:0>, 8) EOR ('0000000':x<63:7>);
Vtmp<127:64> = w<127:64> + sig0;
V[d, 128] = Vtmp;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

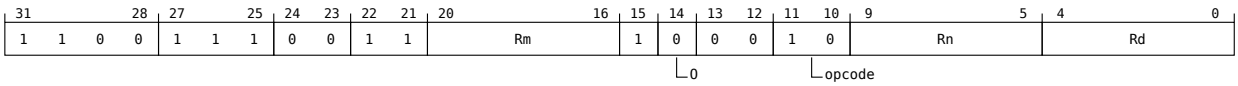
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.237 SHA512SU1

SHA512 schedule update 1

This instruction takes the values from the three source SIMD&FP registers and produces a 128-bit output value that combines the gamma1 functions of two iterations of the SHA512 schedule update that are performed after the first 16 iterations within a block. It returns this value to the destination SIMD&FP register.

Advanced SIMD
(FEAT_SHA512)



Encoding

```
SHA512SU1 <Vd>.2D, <Vn>.2D, <Vm>.2D
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA512) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.

<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(64) sig1;
bits(128) Vtmp;
constant bits(128) x = V[n, 128];
constant bits(128) y = V[m, 128];
constant bits(128) w = V[d, 128];

sig1 = ROR(x<127:64>, 19) EOR ROR(x<127:64>, 61) EOR ('000000':x<127:70>);
Vtmp<127:64> = w<127:64> + sig1 + y<127:64>;
sig1 = ROR(x<63:0>, 19) EOR ROR(x<63:0>, 61) EOR ('000000':x<63:6>);
Vtmp<63:0> = w<63:0> + sig1 + y<63:0>;
V[d, 128] = Vtmp;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.238 SHADD

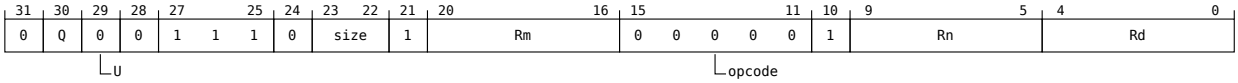
Signed halving add

This instruction adds corresponding signed integer values from the two source SIMD&FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD&FP register.

The results are truncated. For rounded results, see SRHADD.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type
(FEAT_AdvSIMD)



Encoding

```
SHADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    sum = (element1 + element2) >> 1;
    Elem[result, e, esize] = sum<esize-1:0>;

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.239 SHL

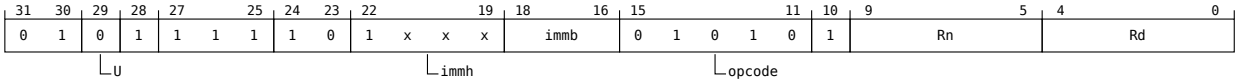
Shift left (immediate)

This instruction reads each value from a vector, left shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



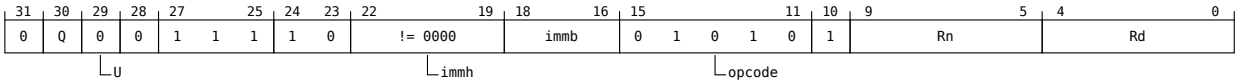
Encoding

```
SHL D<d>, D<n>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh<3> != '1' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << 3;
constant integer datasize = esize;
constant integer elements = 1;
constant integer shift = UInt(immh:immb) - esize;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
SHL <Vd>.<T>, <Vn>.<T>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh<3>:Q == '10' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << HighestSetBitNZ(immh);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
constant integer shift = UInt(immh:immb) - esize;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**n**> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<**shift**> For the "Scalar" variant: is the left shift amount, in the range 0 to 63, encoded as UInt("immh:immb") - 64.

For the "Vector" variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in "immh:immb":

immh	<shift>
0001	UInt(immh:immb) - 8
001x	UInt(immh:immb) - 16
01xx	UInt(immh:immb) - 32
1xxx	UInt(immh:immb) - 64

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> Is an arrangement specifier, encoded in "immh:Q":

immh	Q	<T>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(datasize) result;

for e = 0 to elements-1
    Elem[result, e, esize] = LSL(Elem[operand, e, esize], shift);

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.240 SHLL, SHLL2

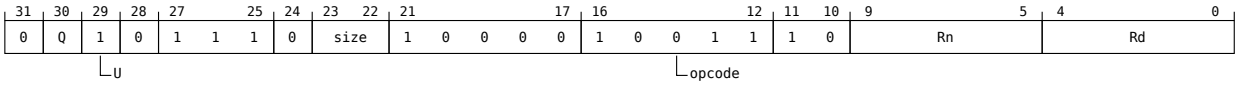
Shift left long (by element size)

This instruction reads each vector element in the lower or upper half of the source SIMD&FP register, left shifts each result by the element size, writes the final result to a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The SHLL instruction extracts vector elements from the lower half of the source register. The SHLL2 instruction extracts vector elements from the upper half of the source register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector
(FEAT_AdvSIMD)



Encoding

```
SHLL{2} <Vd>.<Ta>, <Vn>.<Tb>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;

constant integer shift = esize;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

Q	2
0	[absent]
1	[present]

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**Ta**> Is an arrangement specifier, encoded in "size":

size	<Ta>
00	8H

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<Ta>
01	4S
10	2D
11	RESERVED

<**Vn**> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<**Tb**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**shift**> Is the left shift amount, which must be equal to the source element width in bits, encoded in “size”:

size	<shift>
00	8
01	16
10	32
11	RESERVED

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = Vpart[n, part, datasize];
bits(2*datasize) result;
integer element;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]) << shift;
    Elem[result, e, 2*esize] = element<2*esize-1:0>;

V[d, 2*datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.241 SHRN, SHRN2

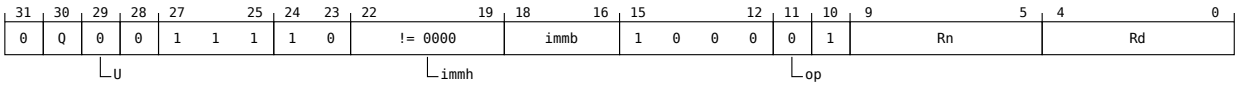
Shift right narrow (immediate)

This instruction reads each unsigned integer value from the source SIMD&FP register, right shifts each result by an immediate value, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements. The results are truncated. For rounded results, see RSHRN.

The SHRN instruction writes the vector to the lower half of the destination register and clears the upper half. The SHRN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector
(FEAT_AdvSIMD)



Encoding

```
SHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh<3> == '1' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << HighestSetBitNZ(immh<2:0>);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;

constant integer shift = (2 * esize) - UInt(immh:immb);
constant boolean round = FALSE;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

Q	2
0	[absent]
1	[present]

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**Tb**> Is an arrangement specifier, encoded in "immh:Q":

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

immh	Q	<Tb>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<**Ta**> Is an arrangement specifier, encoded in "immh":

immh	<Ta>
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

<**shift**> Is the right shift amount, in the range 1 to the destination element width in bits, encoded in "immh:immb":

immh	<shift>
0001	16 - UInt(immh:immb)
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	RESERVED

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize*2) operand = V[n, datasize*2];
bits(datasize) result;
integer element;

for e = 0 to elements-1
    element = RShr(UInt(Elem[operand, e, 2*esize]), shift, round);
    Elem[result, e, esize] = element<esize-1:0>;

Vpart[d, part, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

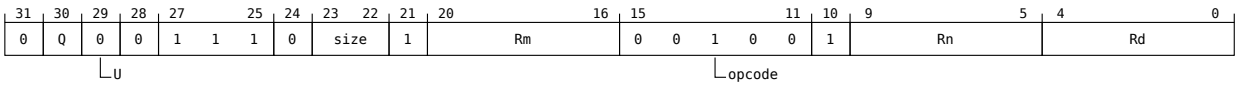
C7.2.242 SHSUB

Signed halving subtract

This instruction subtracts the elements in the vector in the second source SIMD&FP register from the corresponding elements in the vector in the first source SIMD&FP register, shifts each result right one bit, places each result into elements of a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type
(FEAT_AdvSIMD)



Encoding

```
SHSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer diff;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    diff = (element1 - element2) >> 1;
    Elem[result, e, esize] = diff<esize-1:0>;

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

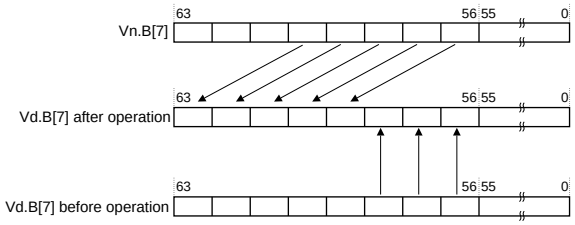
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.243 SLI

Shift left and insert (immediate)

This instruction reads each vector element in the source SIMD&FP register, left shifts each vector element by an immediate value, and inserts the result into the corresponding vector element in the destination SIMD&FP register such that the new zero bits created by the shift are not inserted but retain their existing value. Bits shifted out of the left of each vector element in the source register are lost.

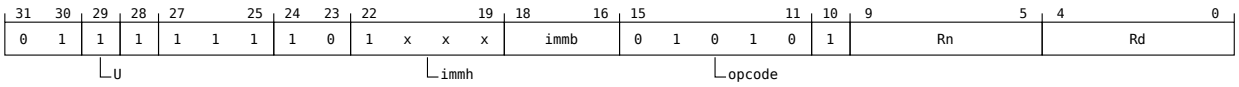
The following figure shows an example of the operation of shift left by 3 for an 8-bit vector element.



Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



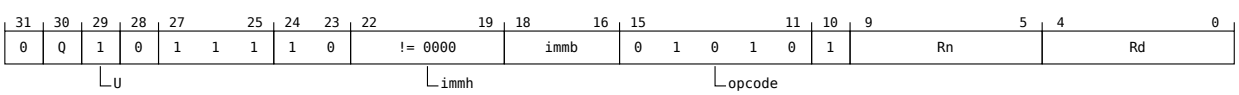
Encoding

```
SLI D<d>, D<n>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh<3> != '1' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << 3;
constant integer datasize = esize;
constant integer elements = 1;
constant integer shift = UInt(immh:immb) - esize;
```

Vector
(FEAT_AdvSIMD)



Encoding

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
SLI <Vd>.<T>, <Vn>.<T>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh<3>:Q == '10' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << HighestSetBitNZ(immh);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
constant integer shift = UInt(immh:immb) - esize;
```

Assembler Symbols

<**d**> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<**n**> Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<**shift**> For the “Scalar” variant: is the left shift amount, in the range 0 to 63, encoded as UInt(“immh:immb”) - 64.

For the “Vector” variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in “immh:immb”:

immh	<shift>
0001	UInt(immh:immb) - 8
001x	UInt(immh:immb) - 16
01xx	UInt(immh:immb) - 32
1xxx	UInt(immh:immb) - 64

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand  = V[n, datasize];
constant bits(datasize) operand2 = V[d, datasize];
constant bits(esize) mask = LSL(Ones(esize), shift);
bits(datasize) result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
bits(esize) shifted;

for e = 0 to elements-1
    shifted = LSL(Elem[operand, e, esize], shift);
    Elem[result, e, esize] = (Elem[operand2, e, esize] AND NOT(mask)) OR shifted;
V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

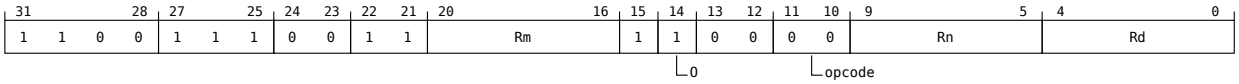
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.244 SM3PARTW1

SM3PARTW1

This instruction takes three 128-bit vectors from the three source SIMD&FP registers and returns a 128-bit result in the destination SIMD&FP register. The result is obtained by a three-way exclusive-OR of the elements within the input vectors with some fixed rotations, see the Operation pseudocode for more information.

Advanced SIMD
(FEAT_SM3)



Encoding

```
SM3PARTW1 <Vd>.4S, <Vn>.4S, <Vm>.4S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SM3) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.

<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(128) Vm = V[m, 128];
constant bits(128) Vn = V[n, 128];
constant bits(128) Vd = V[d, 128];
bits(128) result;

result<95:0> = (Vd EOR Vn)<95:0> EOR (ROL(Vm<127:96>, 15):ROL(Vm<95:64>, 15):ROL(Vm<63:32>, 15));

for i = 0 to 3
    if i == 3 then
        result<127:96> = (Vd EOR Vn)<127:96> EOR (ROL(result<31:0>, 15));
    result<(32*i)+31:(32*i)> = (result<(32*i)+31:(32*i)> EOR ROL(result<(32*i)+31:(32*i)>, 15) EOR
                                ROL(result<(32*i)+31:(32*i)>, 23));
V[d, 128] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

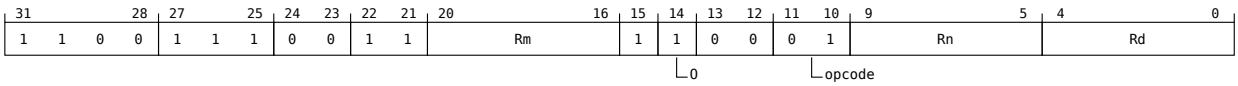
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.245 SM3PARTW2

SM3PARTW2

This instruction takes three 128-bit vectors from three source SIMD&FP registers and returns a 128-bit result in the destination SIMD&FP register. The result is obtained by a three-way exclusive-OR of the elements within the input vectors with some fixed rotations, see the Operation pseudocode for more information.

Advanced SIMD
(FEAT_SM3)



Encoding

```
SM3PARTW2 <Vd>.4S, <Vn>.4S, <Vm>.4S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SM3) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.

<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(128) Vm = V[m, 128];
constant bits(128) Vn = V[n, 128];
constant bits(128) Vd = V[d, 128];

bits(128) result;
bits(128) tmp;
bits(32) tmp2;

tmp<127:0> = Vn EOR (ROL(Vm<127:96>, 7) : ROL(Vm<95:64>, 7) : ROL(Vm<63:32>, 7) : ROL(Vm<31:0>, 7));
result<127:0> = Vd<127:0> EOR tmp<127:0>;
tmp2 = ROL(tmp<31:0>, 15);
tmp2 = tmp2 EOR ROL(tmp2, 15) EOR ROL(tmp2, 23);
result<127:96> = result<127:96> EOR tmp2;
V[d, 128] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

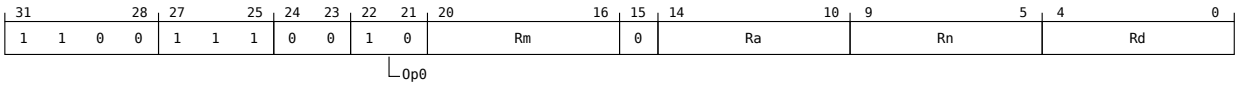
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.246 SM3SS1

SM3SS1

This instruction rotates the top 32 bits of the 128-bit vector in the first source SIMD&FP register by 12, and adds that 32-bit value to the two other 32-bit values held in the top 32 bits of each of the 128-bit vectors in the second and third source SIMD&FP registers, rotating this result left by 7 and writing the final result into the top 32 bits of the vector in the destination SIMD&FP register, with the bottom 96 bits of the vector being written to 0.

Advanced SIMD
(FEAT_SM3)



Encoding

```
SM3SS1 <Vd>.4S, <Vn>.4S, <Vm>.4S, <Va>.4S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SM3) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer a = UInt(Ra);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

<Va> Is the name of the third SIMD&FP source register, encoded in the “Ra” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(128) Vm = V[m, 128];
constant bits(128) Vn = V[n, 128];
constant bits(128) Va = V[a, 128];

bits(128) result;
result<127:96> = ROL((ROL(Vn<127:96>, 12) + Vm<127:96> + Va<127:96>), 7);
result<95:0> = Zeros(96);
V[d, 128] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.247 SM3TT1A

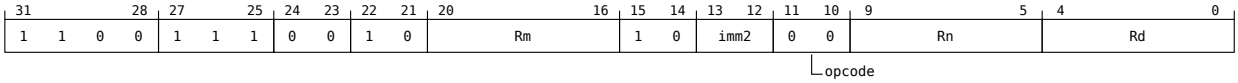
SM3TT1A

This instruction takes three 128-bit vectors from three source SIMD&FP registers and a 2-bit immediate index value, and returns a 128-bit result in the destination SIMD&FP register. It performs a three-way exclusive-OR of the three 32-bit fields held in the upper three elements of the first source vector, and adds the resulting 32-bit value and the following three other 32-bit values:

- The bottom 32-bit element of the first source vector, Vd, that was used for the three-way exclusive-OR.
- The result of the exclusive-OR of the top 32-bit element of the second source vector, Vn, with a rotation left by 12 of the top 32-bit element of the first source vector.
- A 32-bit element indexed out of the third source vector, Vm.

The result of this addition is returned as the top element of the result. The other elements of the result are taken from elements of the first source vector, with the element returned in bits<63:32> being rotated left by 9.

Advanced SIMD
(FEAT_SM3)



Encoding

```
SM3TT1A <Vd>.4S, <Vn>.4S, <Vm>.S[<imm2>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SM3) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer i = UInt(imm2);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the “Rd” field.

<Vn> Is the name of the second SIMD&FP source register, encoded in the “Rn” field.

<Vm> Is the name of the third SIMD&FP source register, encoded in the “Rm” field.

<imm2> Is a 32-bit element indexed out of <Vm>, encoded in “imm2”.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(128) Vm = V[m, 128];
constant bits(128) Vn = V[n, 128];
constant bits(128) Vd = V[d, 128];

bits(32) WjPrime;
bits(128) result;
bits(32) TT1;
bits(32) SS2;

WjPrime = Elem[Vm, i, 32];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
SS2 = Vn<127:96> EOR ROL(Vd<127:96>, 12);
TT1 = Vd<63:32> EOR (Vd<127:96> EOR Vd<95:64>);
TT1 = (TT1 + Vd<31:0> + SS2 + WjPrime)<31:0>;
result<31:0> = Vd<63:32>;
result<63:32> = ROL(Vd<95:64>, 9);
result<95:64> = Vd<127:96>;
result<127:96> = TT1;
V[d, 128] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.248 SM3TT1B

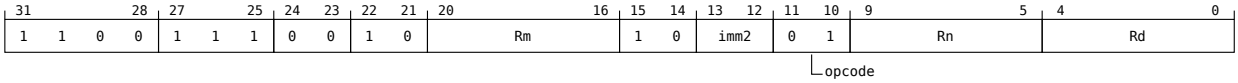
SM3TT1B

This instruction takes three 128-bit vectors from three source SIMD&FP registers and a 2-bit immediate index value, and returns a 128-bit result in the destination SIMD&FP register. It performs a 32-bit majority function between the three 32-bit fields held in the upper three elements of the first source vector, and adds the resulting 32-bit value and the following three other 32-bit values:

- The bottom 32-bit element of the first source vector, Vd, that was used for the 32-bit majority function.
- The result of the exclusive-OR of the top 32-bit element of the second source vector, Vn, with a rotation left by 12 of the top 32-bit element of the first source vector.
- A 32-bit element indexed out of the third source vector, Vm.

The result of this addition is returned as the top element of the result. The other elements of the result are taken from elements of the first source vector, with the element returned in bits<63:32> being rotated left by 9.

Advanced SIMD
(FEAT_SM3)



Encoding

```
SM3TT1B <Vd>.4S, <Vn>.4S, <Vm>.S[<imm2>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SM3) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer i = UInt(imm2);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the “Rd” field.

<Vn> Is the name of the second SIMD&FP source register, encoded in the “Rn” field.

<Vm> Is the name of the third SIMD&FP source register, encoded in the “Rm” field.

<imm2> Is a 32-bit element indexed out of <Vm>, encoded in “imm2”.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(128) Vm = V[m, 128];
constant bits(128) Vn = V[n, 128];
constant bits(128) Vd = V[d, 128];

bits(32) WjPrime;
bits(128) result;
bits(32) TT1;
bits(32) SS2;

WjPrime = Elem[Vm, i, 32];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
SS2 = Vn<127:96> EOR ROL(Vd<127:96>, 12);
TT1 = (Vd<127:96> AND Vd<63:32>) OR (Vd<127:96> AND Vd<95:64>) OR (Vd<63:32> AND Vd<95:64>);
TT1 = (TT1 + Vd<31:0> + SS2 + WjPrime)<31:0>;
result<31:0> = Vd<63:32>;
result<63:32> = ROL(Vd<95:64>, 9);
result<95:64> = Vd<127:96>;
result<127:96> = TT1;
V[d, 128] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.249 SM3TT2A

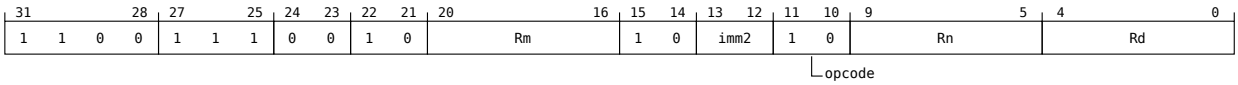
SM3TT2A

This instruction takes three 128-bit vectors from three source SIMD&FP register and a 2-bit immediate index value, and returns a 128-bit result in the destination SIMD&FP register. It performs a three-way exclusive-OR of the three 32-bit fields held in the upper three elements of the first source vector, and adds the resulting 32-bit value and the following three other 32-bit values:

- The bottom 32-bit element of the first source vector, Vd, that was used for the three-way exclusive-OR.
- The 32-bit element held in the top 32 bits of the second source vector, Vn.
- A 32-bit element indexed out of the third source vector, Vm.

A three-way exclusive-OR is performed of the result of this addition, the result of the addition rotated left by 9, and the result of the addition rotated left by 17. The result of this exclusive-OR is returned as the top element of the returned result. The other elements of this result are taken from elements of the first source vector, with the element returned in bits<63:32> being rotated left by 19.

Advanced SIMD
(FEAT_SM3)



Encoding

```
SM3TT2A <Vd>.4S, <Vn>.4S, <Vm>.S[<imm2>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SM3) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer i = UInt(imm2);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the “Rd” field.

<Vn> Is the name of the second SIMD&FP source register, encoded in the “Rn” field.

<Vm> Is the name of the third SIMD&FP source register, encoded in the “Rm” field.

<imm2> Is a 32-bit element indexed out of <Vm>, encoded in “imm2”.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(128) Vm = V[m, 128];
constant bits(128) Vn = V[n, 128];
constant bits(128) Vd = V[d, 128];

bits(32) Wj;
bits(128) result;
bits(32) TT2;

Wj = Elem[Vm, i, 32];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TT2 = Vd<63:32> EOR (Vd<127:96> EOR Vd<95:64>);
TT2 = (TT2 + Vd<31:0> + Vn<127:96> + Wj)<31:0>;

result<31:0> = Vd<63:32>;
result<63:32> = ROL(Vd<95:64>, 19);
result<95:64> = Vd<127:96>;
result<127:96> = TT2 EOR ROL(TT2, 9) EOR ROL(TT2, 17);
V[d, 128] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.250 SM3TT2B

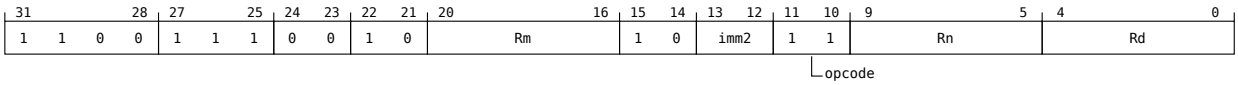
SM3TT2B

This instruction takes three 128-bit vectors from three source SIMD&FP registers, and a 2-bit immediate index value, and returns a 128-bit result in the destination SIMD&FP register. It performs a 32-bit majority function between the three 32-bit fields held in the upper three elements of the first source vector, and adds the resulting 32-bit value and the following three other 32-bit values:

- The bottom 32-bit element of the first source vector, Vd, that was used for the 32-bit majority function.
- The 32-bit element held in the top 32 bits of the second source vector, Vn.
- A 32-bit element indexed out of the third source vector, Vm.

A three-way exclusive-OR is performed of the result of this addition, the result of the addition rotated left by 9, and the result of the addition rotated left by 17. The result of this exclusive-OR is returned as the top element of the returned result. The other elements of this result are taken from elements of the first source vector, with the element returned in bits<63:32> being rotated left by 19.

Advanced SIMD
(FEAT_SM3)



Encoding

```
SM3TT2B <Vd>.4S, <Vn>.4S, <Vm>.S[<imm2>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SM3) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer i = UInt(imm2);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the “Rd” field.

<Vn> Is the name of the second SIMD&FP source register, encoded in the “Rn” field.

<Vm> Is the name of the third SIMD&FP source register, encoded in the “Rm” field.

<imm2> Is a 32-bit element indexed out of <Vm>, encoded in “imm2”.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(128) Vm = V[m, 128];
constant bits(128) Vn = V[n, 128];
constant bits(128) Vd = V[d, 128];

bits(32) Wj;
bits(128) result;
bits(32) TT2;

Wj = Elem[Vm, i, 32];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TT2 = (Vd<127:96> AND Vd<95:64>) OR (NOT(Vd<127:96>) AND Vd<63:32>);
TT2 = (TT2 + Vd<31:0> + Vn<127:96> + Wj)<31:0>;

result<31:0> = Vd<63:32>;
result<63:32> = ROL(Vd<95:64>, 19);
result<95:64> = Vd<127:96>;
result<127:96> = TT2 EOR ROL(TT2, 9) EOR ROL(TT2, 17);
V[d, 128] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

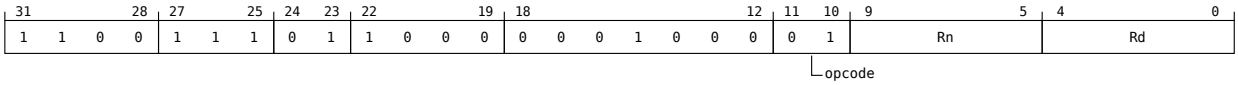
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.251 SM4E

SM4 encode

This instruction takes input data as a 128-bit vector from the first source SIMD&FP register, and four iterations of the round key held as the elements of the 128-bit vector in the second source SIMD&FP register. It encrypts the data by four rounds, in accordance with the SM4 standard, returning the 128-bit result to the destination SIMD&FP register.

Advanced SIMD
(FEAT_SM4)



Encoding

```
SM4E <Vd>.4S, <Vn>.4S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SM4) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.

<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(128) Vn = V[n, 128];
bits(32) intval;
bits(128) roundresult;
bits(32) roundkey;

roundresult = V[d, 128];
for index = 0 to 3
    roundkey = Elem[Vn, index, 32];

    intval = roundresult<127:96> EOR roundresult<95:64> EOR roundresult<63:32> EOR roundkey;

    for i = 0 to 3
        Elem[intval, i, 8]  = Sbox(Elem[intval, i, 8]);

    intval = intval EOR ROL(intval, 2) EOR ROL(intval, 10) EOR ROL(intval, 18) EOR ROL(intval, 24);
    intval = intval EOR roundresult<31:0>;

    roundresult<31:0>   = roundresult<63:32>;
    roundresult<63:32>  = roundresult<95:64>;
    roundresult<95:64>  = roundresult<127:96>;
    roundresult<127:96> = intval;

V[d, 128] = roundresult;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

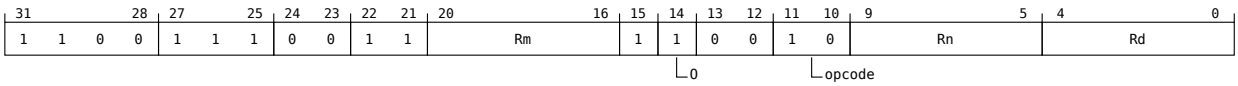
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.252 SM4EKEY

SM4 key

This instruction takes an input as a 128-bit vector from the first source SIMD&FP register and a 128-bit constant from the second SIMD&FP register. It derives four iterations of the output key, in accordance with the SM4 standard, returning the 128-bit result to the destination SIMD&FP register.

Advanced SIMD
(FEAT_SM4)



Encoding

```
SM4EKEY <Vd>.4S, <Vn>.4S, <Vm>.4S
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SM4) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(128) Vm = V[m, 128];
bits(32) intval;
bits(32) const;
bits(128) roundresult;

roundresult = V[n, 128];
for index = 0 to 3
    const = Elem[Vm, index, 32];

    intval = roundresult<127:96> EOR roundresult<95:64> EOR roundresult<63:32> EOR const;

    for i = 0 to 3
        Elem[intval, i, 8] = Sbox(Elem[intval, i, 8]);

    intval = intval EOR ROL(intval, 13) EOR ROL(intval, 23);
    intval = intval EOR roundresult<31:0>;

    roundresult<31:0>   = roundresult<63:32>;
    roundresult<63:32>  = roundresult<95:64>;
    roundresult<95:64>  = roundresult<127:96>;
    roundresult<127:96> = intval;

V[d, 128] = roundresult;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

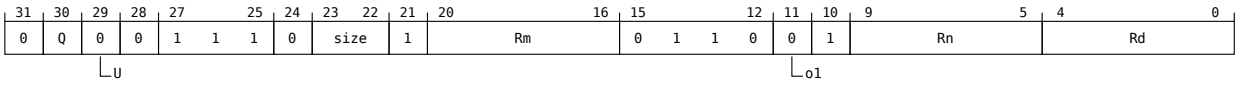
C7.2.253 SMAX

Signed maximum (vector)

This instruction compares corresponding elements in the vectors in the two source SIMD&FP registers, places the larger of each pair of signed integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type
(FEAT_AdvSIMD)



Encoding

```
SMAX <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer max;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    max = Max(element1, element2);
    Elem[result, e, esize] = max<esize-1:0>;

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

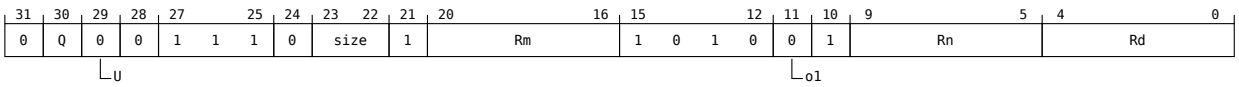
C7.2.254 SMAXP

Signed maximum pairwise

This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the largest of each pair of signed integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type
(FEAT_AdvSIMD)



Encoding

```
SMAXP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
constant bits(2*datasize) concat = operand2:operand1;
integer element1;
integer element2;
integer max;

for e = 0 to elements-1
    element1 = SInt(Elem[concat, 2*e, esize]);
    element2 = SInt(Elem[concat, (2*e)+1, esize]);
    max = Max(element1, element2);
    Elem[result, e, esize] = max<esize-1:0>;

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

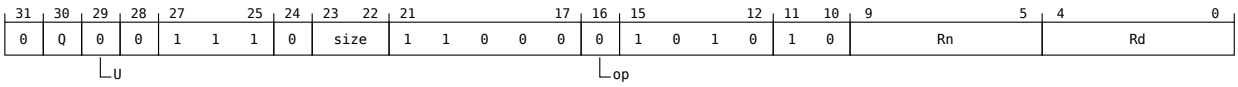
C7.2.255 SMAXV

Signed maximum across vector

This instruction compares all the vector elements in the source SIMD&FP register, and writes the largest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are signed integer values.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Advanced SIMD
(FEAT_AdvSIMD)



Encoding

```
SMAXV <V><d>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '100' then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;

constant boolean unsigned = FALSE;
```

Assembler Symbols

<**V**> Is the destination width specifier, encoded in "size":

size	<V>
00	B
01	H
10	S
11	RESERVED

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<**T**> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	Q	<T>
00	1	16B
01	0	4H
01	1	8H
10	0	RESERVED
10	1	4S
11	x	RESERVED

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(esize) opelt = Elem[operand, 0, esize];
integer max = if unsigned then UInt(opelt) else SInt(opelt);

for e = 1 to elements-1
    opelt = Elem[operand, e, esize];
    constant integer element = if unsigned then UInt(opelt) else SInt(opelt);
    max = Max(max, element);

V[d, esize] = max<esize-1:0>;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

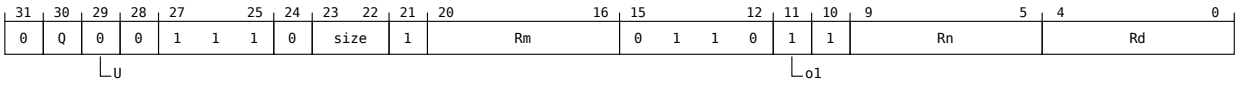
C7.2.256 SMIN

Signed minimum (vector)

This instruction compares corresponding elements in the vectors in the two source SIMD&FP registers, places the smaller of each of the two signed integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type
(FEAT_AdvSIMD)



Encoding

```
SMIN <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer min;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    min = Min(element1, element2);
    Elem[result, e, esize] = min<esize-1:0>;

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

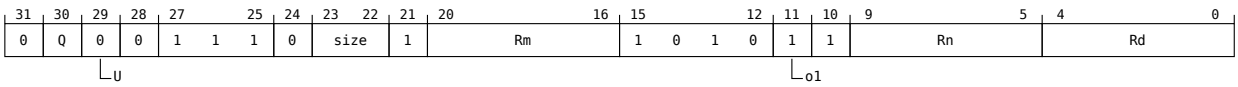
C7.2.257 SMINP

Signed minimum pairwise

This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the smallest of each pair of signed integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type
(FEAT_AdvSIMD)



Encoding

```
SMINP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
constant bits(2*datasize) concat = operand2:operand1;
integer element1;
integer element2;
integer min;

for e = 0 to elements-1
    element1 = SInt(Elem[concat, 2*e, esize]);
    element2 = SInt(Elem[concat, (2*e)+1, esize]);
    min = Min(element1, element2);
    Elem[result, e, esize] = min<esize-1:0>;

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

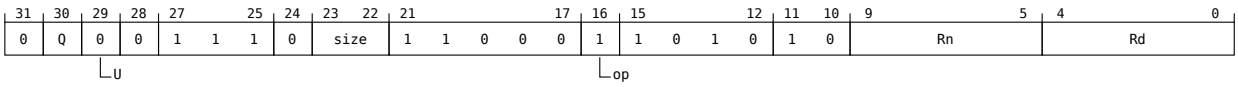
C7.2.258 SMINV

Signed minimum across vector

This instruction compares all the vector elements in the source SIMD&FP register, and writes the smallest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are signed integer values.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Advanced SIMD
(FEAT_AdvSIMD)



Encoding

```
SMINV <V><d>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '100' then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;

constant boolean unsigned = FALSE;
```

Assembler Symbols

<**V**> Is the destination width specifier, encoded in "size":

size	<V>
00	B
01	H
10	S
11	RESERVED

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<**T**> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	Q	<T>
00	1	16B
01	0	4H
01	1	8H
10	0	RESERVED
10	1	4S
11	x	RESERVED

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(esize) opelt = Elem[operand, 0, esize];
integer min = if unsigned then UInt(opelt) else SInt(opelt);

for e = 1 to elements-1
    opelt = Elem[operand, e, esize];
    constant integer element = if unsigned then UInt(opelt) else SInt(opelt);
    min = Min(min, element);

V[d, esize] = min<esize-1:0>;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.259 SMLAL, SMLAL2 (by element)

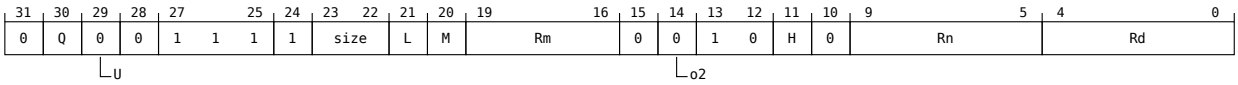
Signed multiply-add long (vector, by element)

This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element in the second source SIMD&FP register, and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are signed integer values.

The `SMLAL` instruction extracts vector elements from the lower half of the first source register. The `SMLAL2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector
(FEAT_AdvSIMD)



Encoding

```
SMLAL{2} <Vd>.<Ta>, <Vn>.<Tb>, V<m>.<Ts>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer idxdsize = 64 << UInt(H);
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rmhi:Rm);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

Q	2
0	[absent]
1	[present]

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**Ta**> Is an arrangement specifier, encoded in "size":

size	<**Ta**>
00	RESERVED
01	4S
10	2D
11	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<**Tb**> Is an arrangement specifier, encoded in "size:Q":

size	**Q**	<**Tb**>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**m**> Is the number of the second SIMD&FP source register, encoded in "size:M:Rm":

size	<**m**>
00	RESERVED
01	UInt('0':Rm)
10	UInt(M:Rm)
11	RESERVED

Restricted to 0-15 when element size <Ts> is H.

<**Ts**> Is an element size specifier, encoded in "size":

size	<**Ts**>
00	RESERVED
01	H
10	S
11	RESERVED

<**index**> Is the element index, encoded in "size:H:L:M":

size	<**index**>
00	RESERVED
01	UInt(H:L:M)

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<index>
10	UInt(H:L)
11	RESERVED

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize)   operand1 = Vpart[n, part, datasize];
constant bits(idxdsize)   operand2 = V[m, idxdsize];
constant bits(2*datasize) operand3 = V[d, 2*datasize];
bits(2*datasize) result;
integer element1;
constant integer element2 = SInt(Elem[operand2, index, esize]);
bits(2*esize) product;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    product = (element1 * element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] + product;

V[d, 2*datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.260 SMLAL, SMLAL2 (vector)

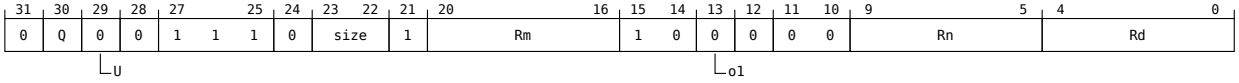
Signed multiply-add long (vector)

This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The SMLAL instruction extracts each source vector from the lower half of each source register. The SMLAL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers, not all the same type
(FEAT_AdvSIMD)



Encoding

```
SMLAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "size":

size	<Ta>
00	8H
01	4S

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<Ta>
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in "size:Q":

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = Vpart[n, part, datasize];
constant bits(datasize) operand2 = Vpart[m, part, datasize];
constant bits(2*datasize) operand3 = V[d, 2*datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
bits(2*esize) accum;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    product = (element1 * element2)<2*esize-1:0>;
    accum = Elem[operand3, e, 2*esize] + product;
    Elem[result, e, 2*esize] = accum;

V[d, 2*datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.261 SMLSL, SMLSL2 (by element)

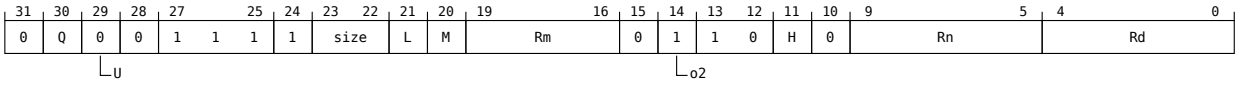
Signed multiply-subtract long (vector, by element)

This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register and subtracts the results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The `SMLSL` instruction extracts vector elements from the lower half of the first source register. The `SMLSL2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector
(FEAT_AdvSIMD)



Encoding

```
SMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, V<m>.<Ts>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer idxdsize = 64 << UInt(H);
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rmhi:Rm);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Ta**> Is an arrangement specifier, encoded in “size”:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Tb**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**m**> Is the number of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<m>
00	RESERVED
01	UInt('0':Rm)
10	UInt(M:Rm)
11	RESERVED

Restricted to 0-15 when element size <Ts> is H.

<**Ts**> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<**index**> Is the element index, encoded in “size:H:L:M”:

size	<index>
00	RESERVED
01	UInt(H:L:M)
10	UInt(H:L)
11	RESERVED

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize)   operand1 = Vpart[n, part, datasize];
constant bits(idxdsize)   operand2 = V[m, idxdsize];
constant bits(2*datasize) operand3 = V[d, 2*datasize];
bits(2*datasize) result;
integer element1;
constant integer element2 = SInt(Elem[operand2, index, esize]);
bits(2*esize) product;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    product = (element1 * element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] - product;

V[d, 2*datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.262 SMLSL, SMLSL2 (vector)

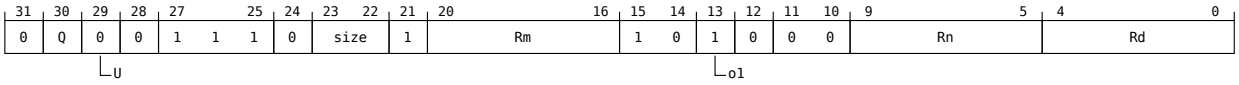
Signed multiply-subtract long (vector)

This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, and subtracts the results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The `SMLSL` instruction extracts each source vector from the lower half of each source register. The `SMLSL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers, not all the same type
(FEAT_AdvSIMD)



Encoding

```
SMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "size":

size	<Ta>
00	8H
01	4S

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<Ta>
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in "size:Q":

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = Vpart[n, part, datasize];
constant bits(datasize) operand2 = Vpart[m, part, datasize];
constant bits(2*datasize) operand3 = V[d, 2*datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
bits(2*esize) accum;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    product = (element1 * element2)<2*esize-1:0>;
    accum = Elem[operand3, e, 2*esize] - product;
    Elem[result, e, 2*esize] = accum;

V[d, 2*datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

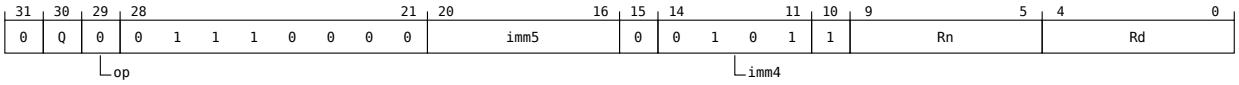
C7.2.263 SMOV

Signed move vector element to general-purpose register

This instruction reads the signed integer from the source SIMD&FP register, sign-extends it to form a 32-bit or 64-bit value, and writes the result to destination general-purpose register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Advanced SIMD
(FEAT_AdvSIMD)



Encoding for the 32-bit variant Applies when `(Q == 0)`

```
SMOV <Wd>, <Vn>.<Ts>[<index>]
```

Encoding for the 64-bit variant Applies when `(Q == 1)`

```
SMOV <Xd>, <Vn>.<Ts>[<index>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if imm5 == 'xx000' then EndOfDecode(Decode_UNDEF);
constant integer size = LowestSetBitNZ(imm5<2:0>);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << size;
constant integer datasize = 32 << UInt(Q);
if datasize <= esize then EndOfDecode(Decode_UNDEF);
constant integer index = UInt(imm5<4:size+1>);
constant integer idxdsize = 64 << UInt(imm5<4>);
```

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Vn**> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<**Ts**> For the “32-bit” variant: is an element size specifier, encoded in “imm5”:

imm5	<Ts>
xxx00	RESERVED
xxxx1	B
xxx10	H

For the “64-bit” variant: is an element size specifier, encoded in “imm5”:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

imm5	<Ts>
xx000	RESERVED
xxxx1	B
xxx10	H
xx100	S

<**index**> For the “32-bit” variant: is the element index encoded in “imm5”:

imm5	<index>
xxx00	RESERVED
xxxx1	UInt(imm5<4:1>)
xxx10	UInt(imm5<4:2>)

For the “64-bit” variant: is the element index encoded in “imm5”:

imm5	<index>
xx000	RESERVED
xxxx1	UInt(imm5<4:1>)
xxx10	UInt(imm5<4:2>)
xx100	UInt(imm5<4:3>)

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(idxdsize) operand = V[n, idxdsize];

X[d, datasize] = SignExtend(Elem[operand, index, esize], datasize);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.264 SMULL, SMULL2 (by element)

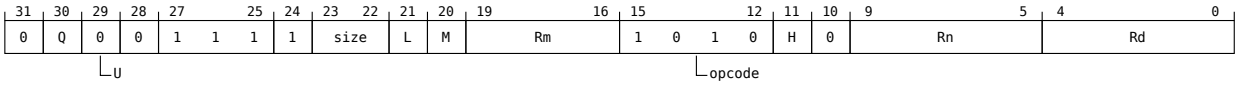
Signed multiply long (vector, by element)

This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The `SMULL` instruction extracts vector elements from the lower half of the first source register. The `SMULL2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector
(FEAT_AdvSIMD)



Encoding

```
SMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, V<m>.<Ts>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer idxdsize = 64 << UInt(H);
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rmhi:Rm);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Ta**> Is an arrangement specifier, encoded in “size”:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Tb**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**m**> Is the number of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<m>
00	RESERVED
01	UInt('0':Rm)
10	UInt(M:Rm)
11	RESERVED

Restricted to 0-15 when element size <Ts> is H.

<**Ts**> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<**index**> Is the element index, encoded in “size:H:L:M”:

size	<index>
00	RESERVED
01	UInt(H:L:M)
10	UInt(H:L)
11	RESERVED

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize)   operand1 = Vpart[n, part, datasize];
constant bits(idxdsize)   operand2 = V[m, idxdsize];
bits(2*datasize) result;
integer element1;
constant integer element2 = SInt(Elem[operand2, index, esize]);
bits(2*esize) product;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    product = (element1 * element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = product;

V[d, 2*datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.265 SMULL, SMULL2 (vector)

Signed multiply long (vector)

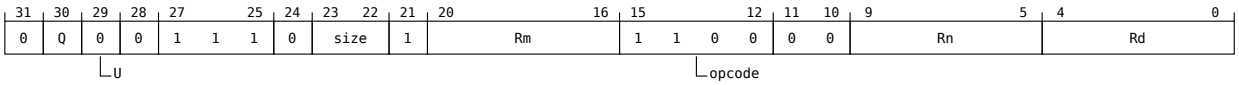
This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, places the results in a vector, and writes the vector to the destination SIMD&FP register.

The destination vector elements are twice as long as the elements that are multiplied.

The SMULL instruction extracts each source vector from the lower half of each source register. The SMULL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers, not all the same type
(FEAT_AdvSIMD)



Encoding

```
SMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "size":

size	<Ta>
00	8H
01	4S

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<Ta>
10	2D
11	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Tb**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize)   operand1 = Vpart[n, part, datasize];
constant bits(datasize)   operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    Elem[result, e, 2*esize] = (element1 * element2)<2*esize-1:0>;

V[d, 2*datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.266 SQABS

Signed saturating absolute value

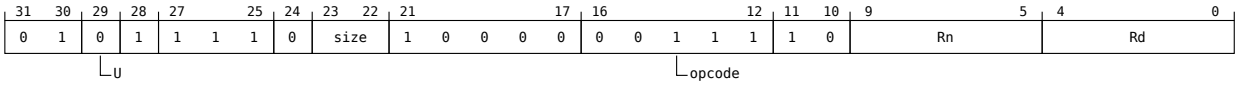
This instruction reads each vector element from the source SIMD&FP register, puts the absolute value of the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



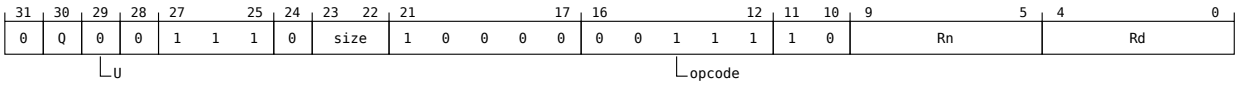
Encoding

```
SQABS <V><d>, <V><n>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
SQABS <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**V**> Is a width specifier, encoded in "size":

size	<V>
00	B
01	H
10	S
11	D

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**n**> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element;
boolean sat;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    element = Abs(element);
    (Elem[result, e, esize], sat) = SignedSatQ(element, esize);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.267 SQADD

Signed saturating add

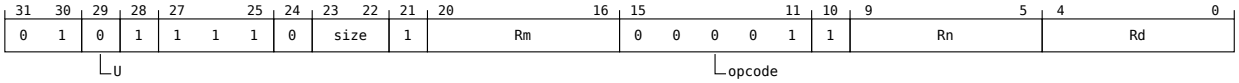
This instruction adds the values of corresponding elements of the two source SIMD&FP registers, places the results into a vector, and writes the vector to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



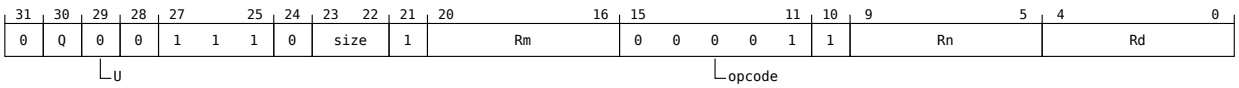
Encoding

```
SQADD <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
constant boolean unsigned = FALSE;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
SQADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
constant boolean unsigned = FALSE;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<**V**> Is a width specifier, encoded in "size":

size	<V>
00	B
01	H
10	S
11	D

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**n**> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<**m**> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer sum;
boolean sat;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    sum = element1 + element2;
    (Elem[result, e, esize], sat) = SatQ(sum, esize, unsigned);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.268 SQDMLAL, SQDMLAL2 (by element)

Signed saturating doubling multiply-add long (by element)

This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, and accumulates the final results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

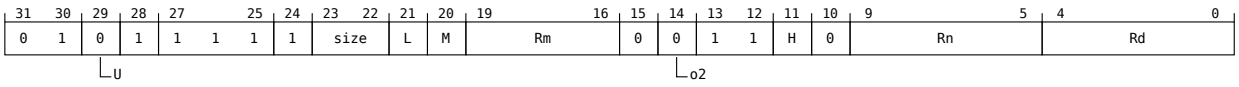
If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The `SQDMLAL` instruction extracts vector elements from the lower half of the first source register. The `SQDMLAL2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



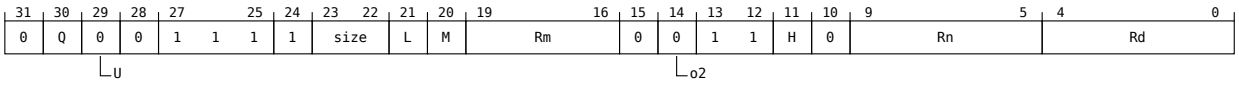
Encoding

`SQDMLAL <Va><d>, <Vb><n>, V<m>.<Ts>[<index>]`

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer idxdsize = 64 << UInt(H);
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rmhi:Rm);

constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
constant integer part = 0;
```

Vector
(FEAT_AdvSIMD)



Encoding

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
SQDMLAL{2} <Vd>.<Ta>, <Vn>.<Tb>, V<m>.<Ts>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer idxdsize = 64 << UInt(H);
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rmhi:Rm);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Va**> Is the destination width specifier, encoded in “size”:

size	<Va>
00	RESERVED
01	S
10	D
11	RESERVED

<**d**> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<**Vb**> Is the source width specifier, encoded in “size”:

size	<Vb>
00	RESERVED
01	H
10	S
11	RESERVED

<**n**> Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<**m**> Is the number of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<m>
00	RESERVED
01	UInt('0':Rm)
10	UInt(M:Rm)
11	RESERVED

Restricted to 0-15 when element size <Ts> is H.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**Ts**> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<**index**> Is the element index, encoded in “size:H:L:M”:

size	<index>
00	RESERVED
01	UInt(H:L:M)
10	UInt(H:L)
11	RESERVED

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Ta**> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Tb**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize)   operand1 = Vpart[n, part, datasize];
constant bits(idxdsize)   operand2 = V[m, idxdsize];
constant bits(2*datasize) operand3 = V[d, 2*datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
integer accum;
boolean sat1;
boolean sat2;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    (product, sat1) = SignedSatQ(2 * element1 * element2, 2 * esize);
    accum = SInt(Elem[operand3, e, 2*esize]) + SInt(product);
    (Elem[result, e, 2*esize], sat2) = SignedSatQ(accum, 2 * esize);
    if sat1 || sat2 then FPSR.QC = '1';

V[d, 2*datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.269 SQDMLAL, SQDMLAL2 (vector)

Signed saturating doubling multiply-add long

This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, doubles the results, and accumulates the final results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

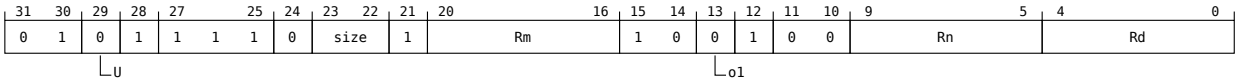
If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The `SQDMLAL` instruction extracts each source vector from the lower half of each source register. The `SQDMLAL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



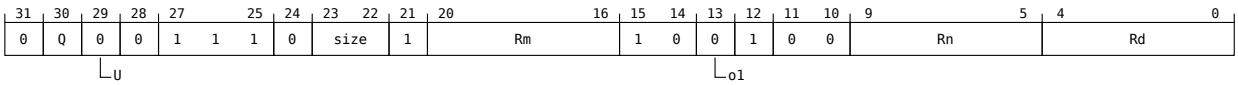
Encoding

```
SQDMLAL <Va><d>, <Vb><n>, <Vb><m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '00' || size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
constant integer part = 0;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
SQDMLAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '00' || size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Va**> Is the destination width specifier, encoded in "size":

size	<Va>
00	RESERVED
01	S
10	D
11	RESERVED

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**Vb**> Is the source width specifier, encoded in "size":

size	<Vb>
00	RESERVED
01	H
10	S
11	RESERVED

<**n**> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<**m**> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

Q	2
0	[absent]
1	[present]

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**Ta**> Is an arrangement specifier, encoded in "size":

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**Tb**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = Vpart[n, part, datasize];
constant bits(datasize) operand2 = Vpart[m, part, datasize];
constant bits(2*datasize) operand3 = V[d, 2*datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
integer accum;
boolean sat1;
boolean sat2;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    (product, sat1) = SignedSatQ(2 * element1 * element2, 2 * esize);
    accum = SInt(Elem[operand3, e, 2*esize]) + SInt(product);
    (Elem[result, e, 2*esize], sat2) = SignedSatQ(accum, 2 * esize);
    if sat1 || sat2 then FPSR.QC = '1';

V[d, 2*datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.270 SQDMLSL, SQDMLSL2 (by element)

Signed saturating doubling multiply-subtract long (by element)

This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, and subtracts the final results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are signed integer values.

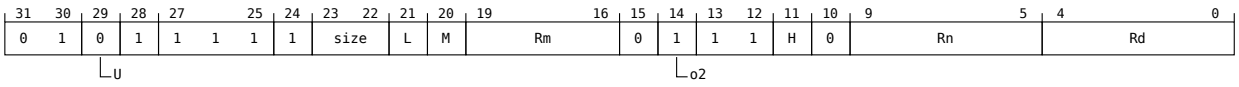
If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The `SQDMLSL` instruction extracts vector elements from the lower half of the first source register. The `SQDMLSL2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



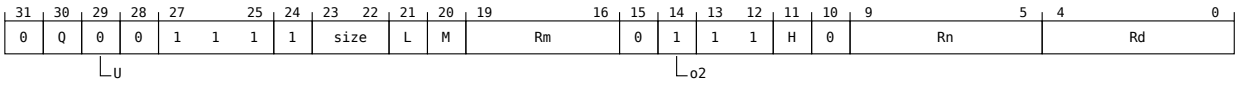
Encoding

```
SQDMLSL <Va><d>, <Vb><n>, V<m>.<Ts>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer idxdsize = 64 << UInt(H);
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rmhi:Rm);

constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
constant integer part = 0;
```

Vector
(FEAT_AdvSIMD)



Encoding

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
SQDMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, V<m>.<Ts>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer idxdsize = 64 << UInt(H);
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rmhi:Rm);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Va**> Is the destination width specifier, encoded in “size”:

size	<Va>
00	RESERVED
01	S
10	D
11	RESERVED

<**d**> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<**Vb**> Is the source width specifier, encoded in “size”:

size	<Vb>
00	RESERVED
01	H
10	S
11	RESERVED

<**n**> Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<**m**> Is the number of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<m>
00	RESERVED
01	UInt('0':Rm)
10	UInt(M:Rm)
11	RESERVED

Restricted to 0-15 when element size <Ts> is H.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**Ts**> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<**index**> Is the element index, encoded in “size:H:L:M”:

size	<index>
00	RESERVED
01	UInt(H:L:M)
10	UInt(H:L)
11	RESERVED

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Ta**> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Tb**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize)   operand1 = Vpart[n, part, datasize];
constant bits(idxdsize)   operand2 = V[m, idxdsize];
constant bits(2*datasize) operand3 = V[d, 2*datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
integer accum;
boolean sat1;
boolean sat2;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    (product, sat1) = SignedSatQ(2 * element1 * element2, 2 * esize);
    accum = SInt(Elem[operand3, e, 2*esize]) - SInt(product);
    (Elem[result, e, 2*esize], sat2) = SignedSatQ(accum, 2 * esize);
    if sat1 || sat2 then FPSR.QC = '1';

V[d, 2*datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.271 SQDMLSL, SQDMLSL2 (vector)

Signed saturating doubling multiply-subtract long

This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, doubles the results, and subtracts the final results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

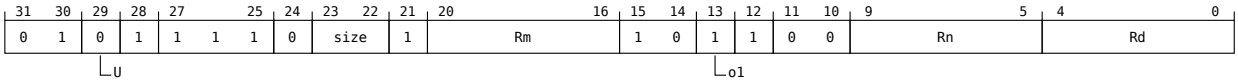
If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The `SQDMLSL` instruction extracts each source vector from the lower half of each source register. The `SQDMLSL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



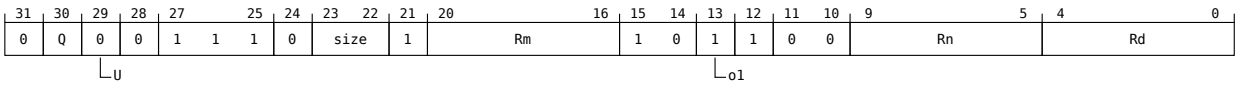
Encoding

```
SQDMLSL <Va><d>, <Vb><n>, <Vb><m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '00' || size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
constant integer part = 0;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
SQDMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '00' || size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Va**> Is the destination width specifier, encoded in "size":

size	<Va>
00	RESERVED
01	S
10	D
11	RESERVED

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**Vb**> Is the source width specifier, encoded in "size":

size	<Vb>
00	RESERVED
01	H
10	S
11	RESERVED

<**n**> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<**m**> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

Q	2
0	[absent]
1	[present]

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**Ta**> Is an arrangement specifier, encoded in "size":

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**Tb**> Is an arrangement specifier, encoded in "size:Q":

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = Vpart[n, part, datasize];
constant bits(datasize) operand2 = Vpart[m, part, datasize];
constant bits(2*datasize) operand3 = V[d, 2*datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
integer accum;
boolean sat1;
boolean sat2;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    (product, sat1) = SignedSatQ(2 * element1 * element2, 2 * esize);
    accum = SInt(Elem[operand3, e, 2*esize]) - SInt(product);
    (Elem[result, e, 2*esize], sat2) = SignedSatQ(accum, 2 * esize);
    if sat1 || sat2 then FPSR.QC = '1';

V[d, 2*datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.272 SQDMULH (by element)

Signed saturating doubling multiply returning high half (by element)

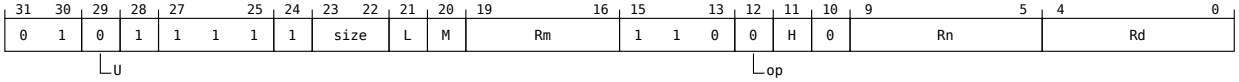
This instruction multiplies each vector element in the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD&FP register.

The results are truncated. For rounded results, see SQRDMULH.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



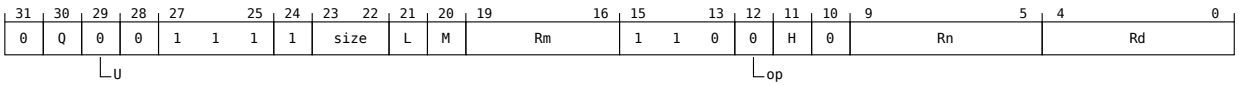
Encoding

```
SQDMULH <V><d>, <V><n>, V<m>.<Ts>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer idxdsize = 64 << UInt(H);
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L);   Rmhi = M;
    otherwise EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rmhi:Rm);

constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;

constant boolean round = FALSE;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
SQDMULH <Vd>.<T>, <Vn>.<T>, V<m>.<Ts>[<index>]
```

Decode for this encoding

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer idxdsize = 64 << UInt(H);
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L);   Rmhi = M;
    otherwise EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rmhi:Rm);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;

constant boolean round = FALSE;
```

Assembler Symbols

<**V**> Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	RESERVED

<**d**> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<**n**> Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<**m**> Is the number of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<m>
00	RESERVED
01	UInt('0':Rm)
10	UInt(M:Rm)
11	RESERVED

Restricted to 0-15 when element size <Ts> is H.

<**Ts**> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**index**> Is the element index, encoded in "size:H:L:M":

size	<index>
00	RESERVED
01	UInt(H:L:M)
10	UInt(H:L)
11	RESERVED

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(idxdsize) operand2 = V[m, idxdsize];
bits(datasize) result;
integer element1;
integer element2;
integer product;
boolean sat;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    product = 2 * element1 * element2;
    product = RShr(product, esize, round);
    // The following only saturates if element1 and element2 equal -(2^(esize-1))
    (Elem[result, e, esize], sat) = SignedSatQ(product, esize);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.273 SQDMULH (vector)

Signed saturating doubling multiply returning high half

This instruction multiplies the values of corresponding elements of the two source SIMD&FP registers, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD&FP register.

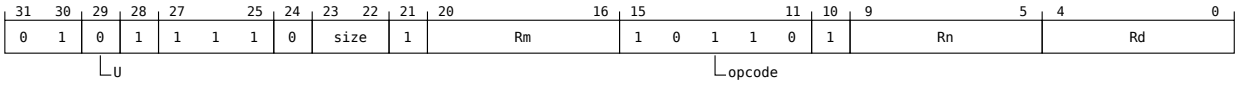
The results are truncated. For rounded results, see SQRDMULH.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



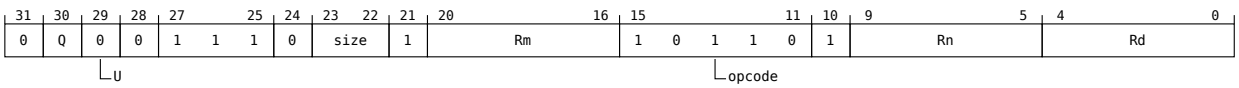
Encoding

```
SQDMULH <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' || size == '00' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
constant boolean rounding = FALSE;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
SQDMULH <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' || size == '00' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
constant boolean rounding = FALSE;
```

Assembler Symbols

<**V**> Is a width specifier, encoded in "size":

size	<V>
00	RESERVED
01	H
10	S
11	RESERVED

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**n**> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<**m**> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer product;
boolean sat;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    product = 2 * element1 * element2;
    product = RShr(product, esize, rounding);
    (Elem[result, e, esize], sat) = SignedSatQ(product, esize);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.274 SQDMULL, SQDMULL2 (by element)

Signed saturating doubling multiply long (by element)

This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, places the final results in a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values.

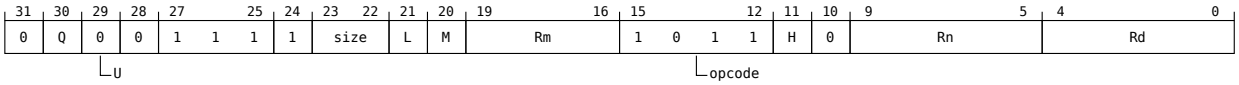
If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The `SQDMULL` instruction extracts the first source vector from the lower half of the first source register. The `SQDMULL2` instruction extracts the first source vector from the upper half of the first source register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Vector and Scalar

Vector
(FEAT_AdvSIMD)



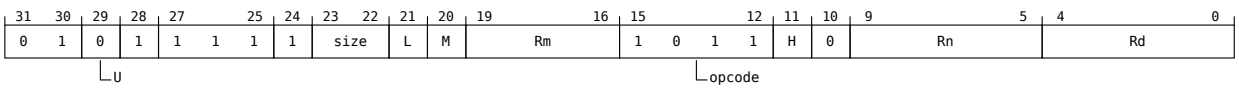
Encoding

```
SQDMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, V<m>.<Ts>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer idxdsize = 64 << UInt(H);
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rmhi:Rm);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;
```

Scalar
(FEAT_AdvSIMD)



Encoding

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
SQDMULL <Va><d>, <Vb><n>, V<m>.<Ts>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer idxdsize = 64 << UInt(H);
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rmhi:Rm);

constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
constant integer part = 0;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Ta**> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Tb**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	Q	<Tb>
11	x	RESERVED

<**m**> Is the number of the second SIMD&FP source register, encoded in "size:M:Rm":

size	<m>
00	RESERVED
01	UInt('0':Rm)
10	UInt(M:Rm)
11	RESERVED

Restricted to 0-15 when element size <Ts> is H.

<**Ts**> Is an element size specifier, encoded in "size":

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<**index**> Is the element index, encoded in "size:H:L:M":

size	<index>
00	RESERVED
01	UInt(H:L:M)
10	UInt(H:L)
11	RESERVED

<**Va**> Is the destination width specifier, encoded in "size":

size	<Va>
00	RESERVED
01	S
10	D
11	RESERVED

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**Vb**> Is the source width specifier, encoded in "size":

size	<Vb>
00	RESERVED

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<Vb>
01	H
10	S
11	RESERVED

<**n**> Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(datasize) operand1 = Vpart[n, part, datasize];
constant bits(idxdsize) operand2 = V[m, idxdsize];
bits(2*datasize) result;
integer element1;
constant integer element2 = SInt(Elem[operand2, index, esize]);
bits(2*esize) product;
boolean sat;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    (product, sat) = SignedSatQ(2 * element1 * element2, 2 * esize);
    Elem[result, e, 2*esize] = product;
    if sat then FPSR.QC = '1';

V[d, 2*datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.275 SQDMULL, SQDMULL2 (vector)

Signed saturating doubling multiply long

This instruction multiplies corresponding vector elements in the lower or upper half of the two source SIMD&FP registers, doubles the results, places the final results in a vector, and writes the vector to the destination SIMD&FP register.

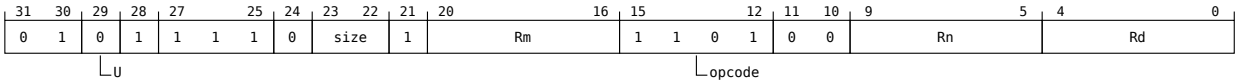
If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The `SQDMULL` instruction extracts each source vector from the lower half of each source register. The `SQDMULL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



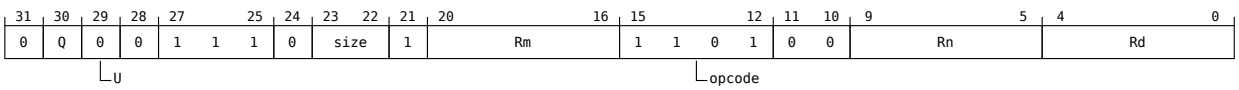
Encoding

```
SQDMULL <Va><d>, <Vb><n>, <Vb><m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '00' || size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
constant integer part = 0;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
SQDMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '00' || size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Va**> Is the destination width specifier, encoded in "size":

size	<Va>
00	RESERVED
01	S
10	D
11	RESERVED

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**Vb**> Is the source width specifier, encoded in "size":

size	<Vb>
00	RESERVED
01	H
10	S
11	RESERVED

<**n**> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<**m**> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

Q	2
0	[absent]
1	[present]

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**Ta**> Is an arrangement specifier, encoded in "size":

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Tb**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = Vpart[n, part, datasize];
constant bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
boolean sat;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    (product, sat) = SignedSatQ(2 * element1 * element2, 2 * esize);
    Elem[result, e, 2*esize] = product;
    if sat then FPSR.QC = '1';

V[d, 2*datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.276 SQNEG

Signed saturating negate

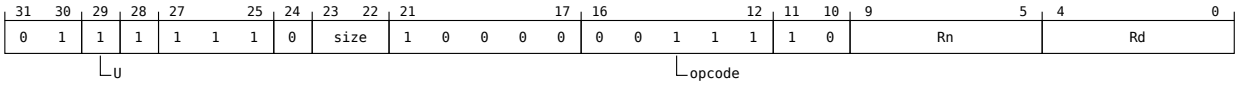
This instruction reads each vector element from the source SIMD&FP register, negates each value, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



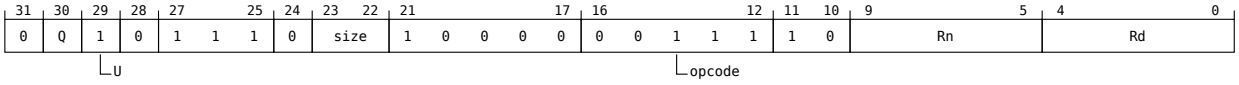
Encoding

```
SQNEG <V><d>, <V><n>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
SQNEG <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**V**> Is a width specifier, encoded in "size":

size	<V>
00	B
01	H
10	S
11	D

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**n**> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element;
boolean sat;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    element = -element;
    (Elem[result, e, esize], sat) = SignedSatQ(element, esize);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.277 SQRDMLAH (by element)

Signed saturating rounding doubling multiply accumulate returning high half (by element)

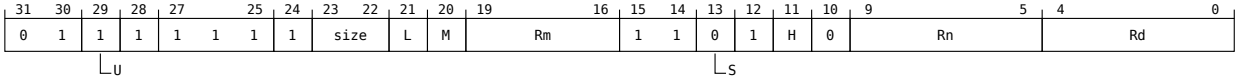
This instruction multiplies the vector elements of the first source SIMD&FP register with the value of a vector element of the second source SIMD&FP register without saturating the multiply results, doubles the results, and accumulates the most significant half of the final results with the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSR.QC, is set if saturation occurs.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_RDM)



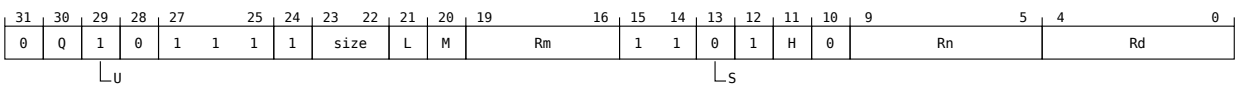
Encoding

```
SQRDMLAH <V><d>, <V><n>, V<m>.<Ts>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_RDM) then EndOfDecode(Decode_UNDEF);
constant integer idxdsize = 64 << UInt(H);
integer index;
bit Rmhi;

case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rmhi:Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;

constant boolean rounding = TRUE;
```

Vector
(FEAT_RDM)



Encoding

```
SQRDMLAH <Vd>.<T>, <Vn>.<T>, V<m>.<Ts>[<index>]
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_RDM) then EndOfDecode(Decode_UNDEF);
constant integer idxdsize = 64 << UInt(H);
integer index;
bit Rmhi;

case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rmhi:Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;

constant boolean rounding = TRUE;
```

Assembler Symbols

<**V**> Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	RESERVED

<**d**> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<**n**> Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<**m**> Is the number of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<m>
00	RESERVED
01	UInt('0':Rm)
10	UInt(M:Rm)
11	RESERVED

Restricted to 0-15 when element size <Ts> is H.

<**Ts**> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**index**> Is the element index, encoded in "size:H:L:M":

size	<index>
00	RESERVED
01	UInt(H:L:M)
10	UInt(H:L)
11	RESERVED

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(idxdsize) operand2 = V[m, idxdsize];
constant bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer element3;
integer accum;
boolean sat;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element3 = SInt(Elem[operand3, e, esize]);
    accum = (element3 << esize) + 2 * (element1 * element2);
    accum = RShr(accum, esize, rounding);
    (Elem[result, e, esize], sat) = SignedSatQ(accum, esize);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.278 SQRDMLAH (vector)

Signed saturating rounding doubling multiply accumulate returning high half (vector)

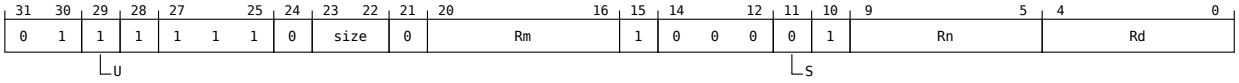
This instruction multiplies the vector elements of the first source SIMD&FP register with the corresponding vector elements of the second source SIMD&FP register without saturating the multiply results, doubles the results, and accumulates the most significant half of the final results with the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSR.QC, is set if saturation occurs.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_RDM)



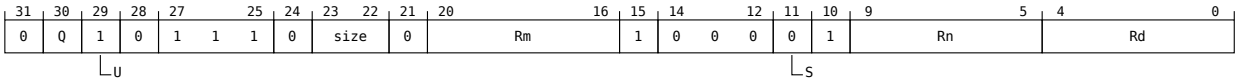
Encoding

```
SQRDMLAH <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_RDM) then EndOfDecode(Decode_UNDEF);
if size == '11' || size == '00' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
constant boolean rounding = TRUE;
```

Vector
(FEAT_RDM)



Encoding

```
SQRDMLAH <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_RDM) then EndOfDecode(Decode_UNDEF);
if size == '11' || size == '00' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer elements = datasize DIV esize;
constant boolean rounding = TRUE;
```

Assembler Symbols

<**V**> Is a width specifier, encoded in "size":

size	<V>
00	RESERVED
01	H
10	S
11	RESERVED

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**n**> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<**m**> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
constant bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer element3;
integer accum;
boolean sat;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    element3 = SInt(Elem[operand3, e, esize]);
    accum = (element3 << esize) + 2 * (element1 * element2);
    accum = RShr(accum, esize, rounding);
    (Elem[result, e, esize], sat) = SignedSatQ(accum, esize);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.279 SQRDMLSH (by element)

Signed saturating rounding doubling multiply subtract returning high half (by element)

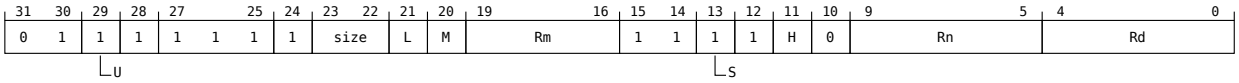
This instruction multiplies the vector elements of the first source SIMD&FP register with the value of a vector element of the second source SIMD&FP register without saturating the multiply results, doubles the results, and subtracts the most significant half of the final results from the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSR.QC, is set if saturation occurs.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_RDM)



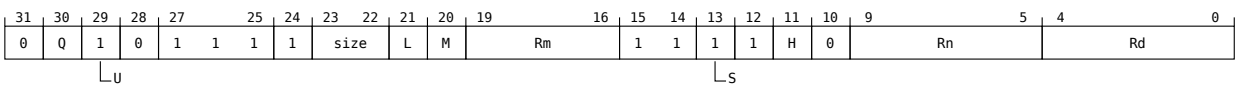
Encoding

```
SQRDMLSH <V><d>, <V><n>, V<m>.<Ts>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_RDM) then EndOfDecode(Decode_UNDEF);
constant integer idxdsize = 64 << UInt(H);
integer index;
bit Rmhi;

case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rmhi:Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;

constant boolean rounding = TRUE;
```

Vector
(FEAT_RDM)



Encoding

```
SQRDMLSH <Vd>.<T>, <Vn>.<T>, V<m>.<Ts>[<index>]
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_RDM) then EndOfDecode(Decode_UNDEF);
constant integer idxdsize = 64 << UInt(H);
integer index;
bit Rmhi;

case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rmhi:Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;

constant boolean rounding = TRUE;
```

Assembler Symbols

<**V**> Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	RESERVED

<**d**> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<**n**> Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<**m**> Is the number of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<m>
00	RESERVED
01	UInt('0':Rm)
10	UInt(M:Rm)
11	RESERVED

Restricted to 0-15 when element size <Ts> is H.

<**Ts**> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**index**> Is the element index, encoded in "size:H:L:M":

size	<index>
00	RESERVED
01	UInt(H:L:M)
10	UInt(H:L)
11	RESERVED

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(idxdsize) operand2 = V[m, idxdsize];
constant bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer element3;
integer accum;
boolean sat;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element3 = SInt(Elem[operand3, e, esize]);
    accum = (element3 << esize) - 2 * (element1 * element2);
    accum = RShr(accum, esize, rounding);
    (Elem[result, e, esize], sat) = SignedSatQ(accum, esize);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.280 SQRDMLSH (vector)

Signed saturating rounding doubling multiply subtract returning high half (vector)

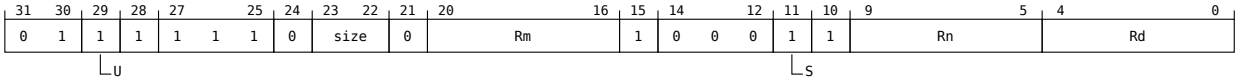
This instruction multiplies the vector elements of the first source SIMD&FP register with the corresponding vector elements of the second source SIMD&FP register without saturating the multiply results, doubles the results, and subtracts the most significant half of the final results from the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSR.QC, is set if saturation occurs.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_RDM)



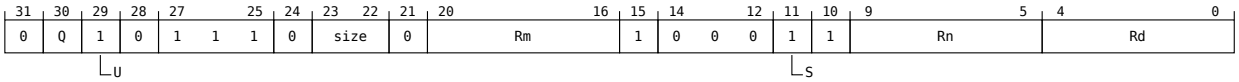
Encoding

```
SQRDMLSH <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_RDM) then EndOfDecode(Decode_UNDEF);
if size == '11' || size == '00' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
constant boolean rounding = TRUE;
```

Vector
(FEAT_RDM)



Encoding

```
SQRDMLSH <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_RDM) then EndOfDecode(Decode_UNDEF);
if size == '11' || size == '00' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer elements = datasize DIV esize;
constant boolean rounding = TRUE;
```

Assembler Symbols

<**V**> Is a width specifier, encoded in "size":

size	<V>
00	RESERVED
01	H
10	S
11	RESERVED

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**n**> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<**m**> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
constant bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer element3;
integer accum;
boolean sat;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    element3 = SInt(Elem[operand3, e, esize]);
    accum = (element3 << esize) - 2 * (element1 * element2);
    accum = RShr(accum, esize, rounding);
    (Elem[result, e, esize], sat) = SignedSatQ(accum, esize);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if sat then FPSR.QC = '1';

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.281 SQRDMULH (by element)

Signed saturating rounding doubling multiply returning high half (by element)

This instruction multiplies each vector element in the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD&FP register.

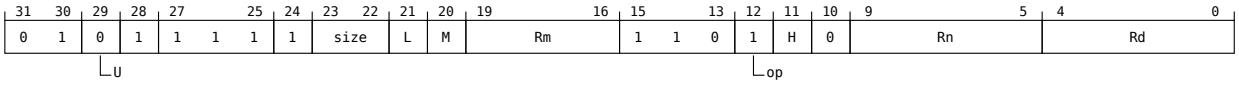
The results are rounded. For truncated results, see SQDMULH.

If any of the results overflows, they are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



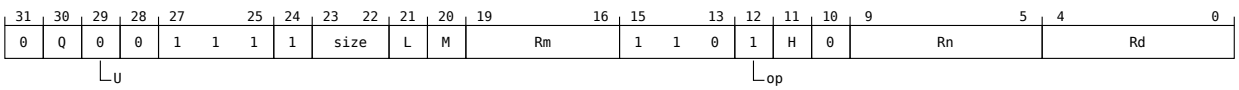
Encoding

```
SQRDMULH <V><d>, <V><n>, V<m>.<Ts>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer idxdsize = 64 << UInt(H);
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L);   Rmhi = M;
    otherwise EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rmhi:Rm);

constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;

constant boolean round = TRUE;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
SQRDMULH <Vd>.<T>, <Vn>.<T>, V<m>.<Ts>[<index>]
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer idxdsize = 64 << UInt(H);
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L);   Rmhi = M;
    otherwise EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rmhi:Rm);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;

constant boolean round = TRUE;
```

Assembler Symbols

<**V**> Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	RESERVED

<**d**> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<**n**> Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<**m**> Is the number of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<m>
00	RESERVED
01	UInt('0':Rm)
10	UInt(M:Rm)
11	RESERVED

Restricted to 0-15 when element size <Ts> is H.

<**Ts**> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**index**> Is the element index, encoded in "size:H:L:M":

size	<index>
00	RESERVED
01	UInt(H:L:M)
10	UInt(H:L)
11	RESERVED

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(idxdsize) operand2 = V[m, idxdsize];
bits(datasize) result;
integer element1;
integer element2;
integer product;
boolean sat;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    product = 2 * element1 * element2;
    product = RShr(product, esize, round);
    // The following only saturates if element1 and element2 equal -(2^(esize-1))
    (Elem[result, e, esize], sat) = SignedSatQ(product, esize);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.282 SQRDMULH (vector)

Signed saturating rounding doubling multiply returning high half

This instruction multiplies the values of corresponding elements of the two source SIMD&FP registers, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD&FP register.

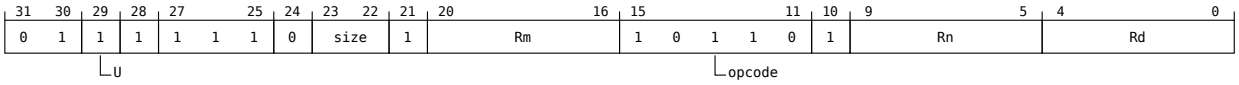
The results are rounded. For truncated results, see SQDMULH.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



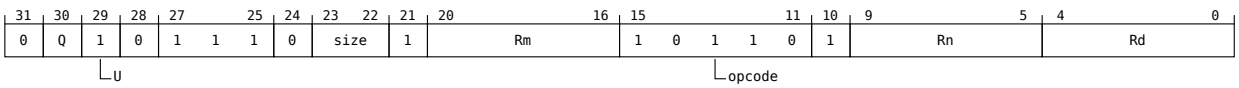
Encoding

```
SQRDMULH <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' || size == '00' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
constant boolean rounding = TRUE;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
SQRDMULH <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' || size == '00' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
constant boolean rounding = TRUE;
```

Assembler Symbols

<**V**> Is a width specifier, encoded in "size":

size	<V>
00	RESERVED
01	H
10	S
11	RESERVED

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**n**> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<**m**> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer product;
boolean sat;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    product = 2 * element1 * element2;
    product = RShr(product, esize, rounding);
    (Elem[result, e, esize], sat) = SignedSatQ(product, esize);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.283 SQRSHL

Signed saturating rounding shift left (register)

This instruction takes each vector element in the first source SIMD&FP register, shifts it by a value from the least significant byte of the corresponding vector element of the second source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

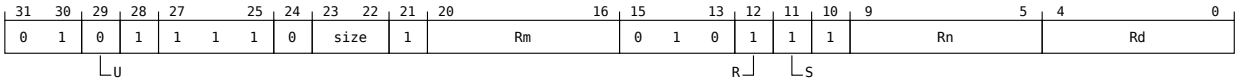
If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are rounded. For truncated results, see SQSHL.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



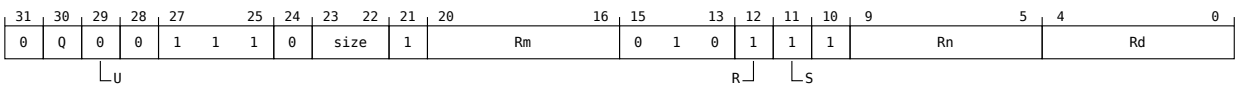
Encoding

```
SQRSHL <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if S == '0' && size != '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant boolean unsigned = FALSE;
constant boolean rounding = TRUE;
constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
SQRSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant boolean unsigned = FALSE;
constant boolean rounding = TRUE;
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**V**> Is a width specifier, encoded in "size":

size	<V>
00	B
01	H
10	S
11	D

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**n**> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<**m**> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
boolean sat;

for e = 0 to elements-1
    integer element = SInt(Elem[operand1, e, esize]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    integer shift = ShiftSat(SInt(Elem[operand2, e, esize]<7:0>), esize);
    if shift >= 0 then // left shift
        element = element << shift;
    else               // right shift
        shift = -shift;
        element = RShr(element, shift, rounding);

    (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.284 SQRSHRN, SQRSHRN2

Signed saturating rounded shift right narrow (immediate)

This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, saturates each shifted result to a value that is half the original width, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are signed integer values. The destination vector elements are half as long as the source vector elements. The results are rounded. For truncated results, see SQSHRN.

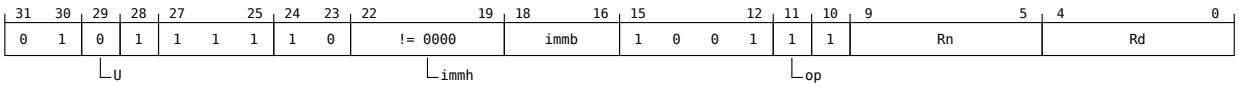
The `SQRSHRN` instruction writes the vector to the lower half of the destination register and clears the upper half. The `SQRSHRN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



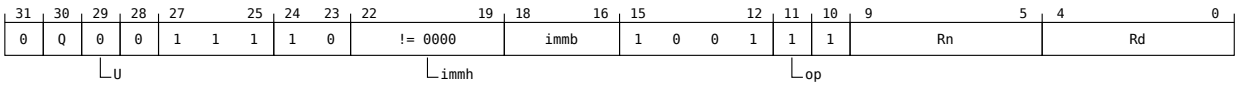
Encoding

```
SQRSHRN <Vb><d>, <Va><n>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh == '0000' then EndOfDecode(Decode_UNDEF);
if immh<3> == '1' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << HighestSetBitNZ(immh<2:0>);
constant integer datasize = esize;
constant integer elements = 1;
constant integer part = 0;

constant integer shift = (2 * esize) - UInt(immh:immb);
constant boolean round = TRUE;
constant boolean unsigned = FALSE;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
SQRSHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh<3> == '1' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << HighestSetBitNZ(immh<2:0>);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;

constant integer shift = (2 * esize) - UInt(immh:immb);
constant boolean round = TRUE;
constant boolean unsigned = FALSE;
```

Assembler Symbols

<**Vb**> Is the destination width specifier, encoded in "immh":

immh	<Vb>
0001	B
001x	H
01xx	S
1xxx	RESERVED

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**Va**> Is the source width specifier, encoded in "immh":

immh	<Va>
0001	H
001x	S
01xx	D
1xxx	RESERVED

<**n**> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<**shift**> For the "Scalar" variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in "immh:immb":

immh	<shift>
0001	16 - UInt(immh:immb)
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	RESERVED

For the "Vector" variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in "immh:immb":

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

immh	<shift>
0001	16 - UInt(immh:immb)
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	RESERVED

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

Q	2
0	[absent]
1	[present]

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**Tb**> Is an arrangement specifier, encoded in "immh:Q":

immh	Q	<Tb>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<**Ta**> Is an arrangement specifier, encoded in "immh":

immh	<Ta>
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize*2) operand = V[n, datasize*2];
bits(datasize) result;
boolean sat;

for e = 0 to elements-1
    constant bits(2*esize) opelt = Elem[operand, e, 2*esize];
    if unsigned then
        constant integer element = RShr(UInt(opelt), shift, round);
        (Elem[result, e, esize], sat) = UnsignedSatQ(element, esize);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            constant integer element = RShr(SInt(opelt), shift, round);
            (Elem[result, e, esize], sat) = SignedSatQ(element, esize);
        if sat then FPSR.QC = '1';

    Vpart[d, part, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.285 SQRSHRUN, SQRSHRUN2

Signed saturating rounded shift right unsigned narrow (immediate)

This instruction reads each signed integer value in the vector of the source SIMD&FP register, right shifts each value by an immediate value, saturates the result to an unsigned integer value that is half the original width, places the final result into a vector, and writes the vector to the destination SIMD&FP register. The results are rounded. For truncated results, see SQSHRUN.

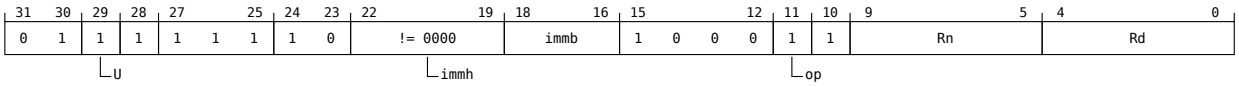
The `SQRSHRUN` instruction writes the vector to the lower half of the destination register and clears the upper half. The `SQRSHRUN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



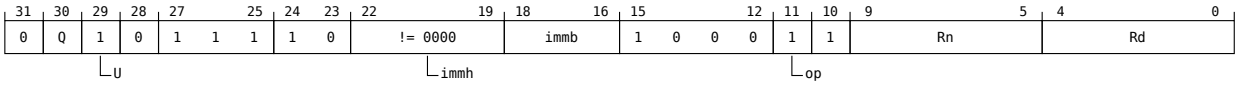
Encoding

```
SQRSHRUN <Vb><d>, <Va><n>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh == '0000' then EndOfDecode(Decode_UNDEF);
if immh<3> == '1' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << HighestSetBitNZ(immh<2:0>);
constant integer datasize = esize;
constant integer elements = 1;
constant integer part = 0;

constant integer shift = (2 * esize) - UInt(immh:immb);
constant boolean round = TRUE;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
SQRSHRUN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
```

Decode for this encoding

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh<3> == '1' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << HighestSetBitNZ(immh<2:0>);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;

constant integer shift = (2 * esize) - UInt(immh:immb);
constant boolean round = TRUE;
```

Assembler Symbols

<**Vb**> Is the destination width specifier, encoded in "immh":

immh	<Vb>
0001	B
001x	H
01xx	S
1xxx	RESERVED

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**Va**> Is the source width specifier, encoded in "immh":

immh	<Va>
0001	H
001x	S
01xx	D
1xxx	RESERVED

<**n**> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<**shift**> For the "Scalar" variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in "immh:immb":

immh	<shift>
0001	16 - UInt(immh:immb)
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	RESERVED

For the "Vector" variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in "immh:immb":

immh	<shift>
0001	16 - UInt(immh:immb)

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

immh	<shift>
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	RESERVED

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

Q	2
0	[absent]
1	[present]

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**Tb**> Is an arrangement specifier, encoded in "immh:Q":

immh	Q	<Tb>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<**Ta**> Is an arrangement specifier, encoded in "immh":

immh	<Ta>
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize*2) operand = V[n, datasize*2];
bits(datasize) result;
integer element;
boolean sat;

for e = 0 to elements-1
    element = RShr(SInt(Elem[operand, e, 2*esize]), shift, round);
    (Elem[result, e, esize], sat) = UnsignedSatQ(element, esize);
    if sat then FPSR.QC = '1';

Vpart[d, part, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.286 SQSHL (immediate)

Signed saturating shift left (immediate)

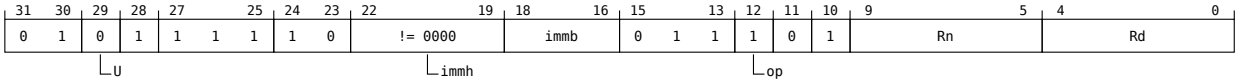
This instruction reads each vector element in the source SIMD&FP register, shifts each result by an immediate value, places the final result in a vector, and writes the vector to the destination SIMD&FP register. The results are truncated. For rounded results, see UQRSHL.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



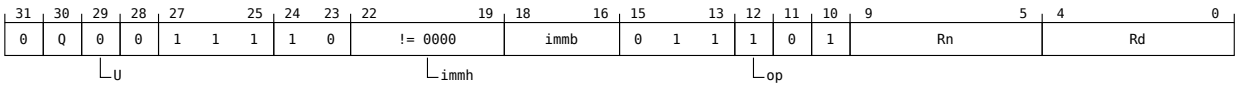
Encoding

```
SQSHL <V><d>, <V><n>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh == '0000' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << HighestSetBitNZ(immh);
constant integer datasize = esize;
constant integer elements = 1;
constant integer shift = UInt(immh:immb) - esize;

constant boolean src_unsigned = FALSE;
constant boolean dst_unsigned = FALSE;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
SQSHL <Vd>.<T>, <Vn>.<T>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh<3>:Q == '10' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer n = UInt(Rn);
constant integer esize = 8 << HighestSetBitNZ(immh);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
constant integer shift = UInt(immh:immb) - esize;

constant boolean src_unsigned = FALSE;
constant boolean dst_unsigned = FALSE;
```

Assembler Symbols

<**V**> Is a width specifier, encoded in “immh”:

immh	<V>
0001	B
001x	H
01xx	S
1xxx	D

<**d**> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<**n**> Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<**shift**> For the “Scalar” variant: is the left shift amount, in the range 0 to the operand width in bits minus 1, encoded in “immh:immb”:

immh	<shift>
0001	UInt(immh:immb) - 8
001x	UInt(immh:immb) - 16
01xx	UInt(immh:immb) - 32
1xxx	UInt(immh:immb) - 64

For the “Vector” variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in “immh:immb”:

immh	<shift>
0001	UInt(immh:immb) - 8
001x	UInt(immh:immb) - 16
01xx	UInt(immh:immb) - 32
1xxx	UInt(immh:immb) - 64

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0001	0	8B
0001	1	16B

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

immh	Q	<T>
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand  = V[n, datasize];
bits(datasize) result;
boolean sat;

for e = 0 to elements-1
    constant bits(esize) opelt = Elem[operand, e, esize];
    constant integer element = if src_unsigned then UInt(opelt) else SInt(opelt);
    (Elem[result, e, esize], sat) = SatQ(element << shift, esize, dst_unsigned);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.287 SQSHL (register)

Signed saturating shift left (register)

This instruction takes each element in the vector of the first source SIMD&FP register, shifts each element by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

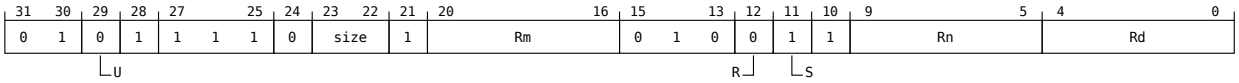
If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are truncated. For rounded results, see SQRSHL.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



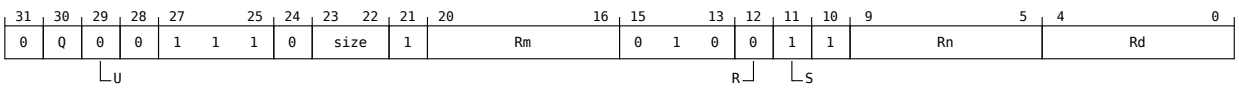
Encoding

```
SQSHL <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if S == '0' && size != '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant boolean unsigned = FALSE;
constant boolean rounding = FALSE;
constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
SQSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant boolean unsigned = FALSE;
constant boolean rounding = FALSE;
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**V**> Is a width specifier, encoded in "size":

size	<V>
00	B
01	H
10	S
11	D

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**n**> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<**m**> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
boolean sat;

for e = 0 to elements-1
    integer element = SInt(Elem[operand1, e, esize]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    integer shift = ShiftSat(SInt(Elem[operand2, e, esize]<7:0>), esize);
    if shift >= 0 then // left shift
        element = element << shift;
    else               // right shift
        shift = -shift;
        element = RShr(element, shift, rounding);

    (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.288 SQSHLU

Signed saturating shift left unsigned (immediate)

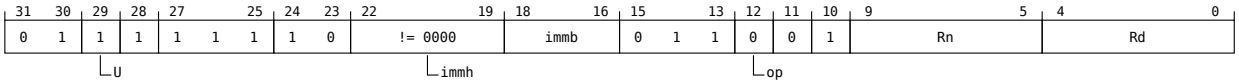
This instruction reads each signed integer value in the vector of the source SIMD&FP register, shifts each value by an immediate value, saturates the shifted result to an unsigned integer value, places the result in a vector, and writes the vector to the destination SIMD&FP register. The results are truncated. For rounded results, see UQRSHL.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



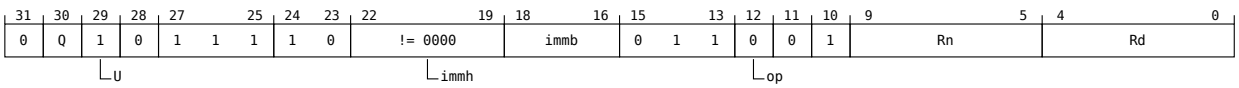
Encoding

```
SQSHLU <V><d>, <V><n>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh == '0000' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << HighestSetBitNZ(immh);
constant integer datasize = esize;
constant integer elements = 1;
constant integer shift = UInt(immh:immb) - esize;

constant boolean src_unsigned = FALSE;
constant boolean dst_unsigned = TRUE;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
SQSHLU <Vd>.<T>, <Vn>.<T>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh<3>:Q == '10' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer esize = 8 << HighestSetBitNZ(immh);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
constant integer shift = UInt(immh:immb) - esize;

constant boolean src_unsigned = FALSE;
constant boolean dst_unsigned = TRUE;
```

Assembler Symbols

<**V**> Is a width specifier, encoded in "immh":

immh	<V>
0001	B
001x	H
01xx	S
1xxx	D

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**n**> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<**shift**> For the "Scalar" variant: is the left shift amount, in the range 0 to the operand width in bits minus 1, encoded in "immh:immb":

immh	<shift>
0001	UInt(immh:immb) - 8
001x	UInt(immh:immb) - 16
01xx	UInt(immh:immb) - 32
1xxx	UInt(immh:immb) - 64

For the "Vector" variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in "immh:immb":

immh	<shift>
0001	UInt(immh:immb) - 8
001x	UInt(immh:immb) - 16
01xx	UInt(immh:immb) - 32
1xxx	UInt(immh:immb) - 64

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> Is an arrangement specifier, encoded in "immh:Q":

immh	Q	<T>
0001	0	8B
0001	1	16B
001x	0	4H

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

immh	Q	<T>
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand  = V[n, datasize];
bits(datasize) result;
boolean sat;

for e = 0 to elements-1
    constant bits(esize) opelt = Elem[operand, e, esize];
    constant integer element = if src_unsigned then UInt(opelt) else SInt(opelt);
    (Elem[result, e, esize], sat) = SatQ(element << shift, esize, dst_unsigned);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.289 SQSHRN, SQSHRN2

Signed saturating shift right narrow (immediate)

This instruction reads each vector element in the source SIMD&FP register, right shifts and truncates each result by an immediate value, saturates each shifted result to a value that is half the original width, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are signed integer values. The destination vector elements are half as long as the source vector elements. For rounded results, see SQRSHRN.

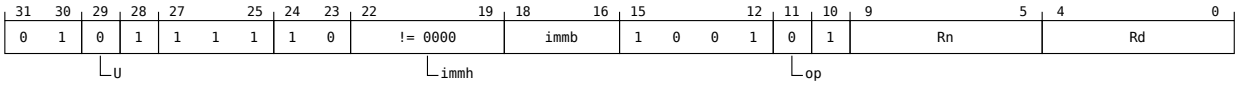
The `SQSHRN` instruction writes the vector to the lower half of the destination register and clears the upper half. The `SQSHRN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



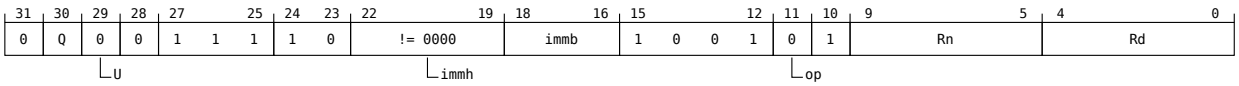
Encoding

```
SQSHRN <Vb><d>, <Va><n>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh == '0000' then EndOfDecode(Decode_UNDEF);
if immh<3> == '1' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << HighestSetBitNZ(immh<2:0>);
constant integer datasize = esize;
constant integer elements = 1;
constant integer part = 0;

constant integer shift = (2 * esize) - UInt(immh:immb);
constant boolean round = FALSE;
constant boolean unsigned = FALSE;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
SQSHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh<3> == '1' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << HighestSetBitNZ(immh<2:0>);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;

constant integer shift = (2 * esize) - UInt(immh:immb);
constant boolean round = FALSE;
constant boolean unsigned = FALSE;
```

Assembler Symbols

<**Vb**> Is the destination width specifier, encoded in “immh”:

immh	<**Vb**>
0001	B
001x	H
01xx	S
1xxx	RESERVED

<**d**> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<**Va**> Is the source width specifier, encoded in “immh”:

immh	<**Va**>
0001	H
001x	S
01xx	D
1xxx	RESERVED

<**n**> Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<**shift**> For the “Scalar” variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in “immh:immb”:

immh	<**shift**>
0001	16 - UInt(immh:immb)
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	RESERVED

For the “Vector” variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immb”:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

immh	<shift>
0001	16 - UInt(immh:immb)
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	RESERVED

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

Q	2
0	[absent]
1	[present]

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**Tb**> Is an arrangement specifier, encoded in "immh:Q":

immh	Q	<Tb>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<**Ta**> Is an arrangement specifier, encoded in "immh":

immh	<Ta>
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize*2) operand = V[n, datasize*2];
bits(datasize) result;
boolean sat;

for e = 0 to elements-1
    constant bits(2*esize) opelt = Elem[operand, e, 2*esize];
    if unsigned then
        constant integer element = RShr(UInt(opelt), shift, round);
        (Elem[result, e, esize], sat) = UnsignedSatQ(element, esize);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        constant integer element = RShr(SInt(opelt), shift, round);
        (Elem[result, e, esize], sat) = SignedSatQ(element, esize);
    if sat then FPSR.QC = '1';

Vpart[d, part, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.290 SQSHRUN, SQSHRUN2

Signed saturating shift right unsigned narrow (immediate)

This instruction reads each signed integer value in the vector of the source SIMD&FP register, right shifts each value by an immediate value, saturates the result to an unsigned integer value that is half the original width, places the final result into a vector, and writes the vector to the destination SIMD&FP register. The results are truncated. For rounded results, see SQRSHRUN.

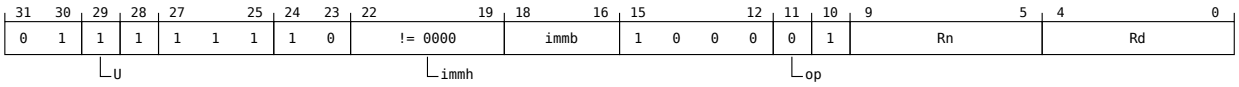
The `SQSHRUN` instruction writes the vector to the lower half of the destination register and clears the upper half. The `SQSHRUN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



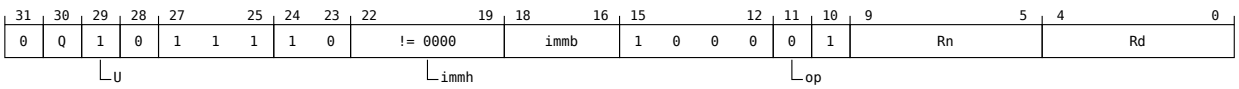
Encoding

```
SQSHRUN <Vb><d>, <Va><n>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh == '0000' then EndOfDecode(Decode_UNDEF);
if immh<3> == '1' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << HighestSetBitNZ(immh<2:0>);
constant integer datasize = esize;
constant integer elements = 1;
constant integer part = 0;

constant integer shift = (2 * esize) - UInt(immh:immb);
constant boolean round = FALSE;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
SQSHRUN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
```

Decode for this encoding

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh<3> == '1' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << HighestSetBitNZ(immh<2:0>);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;

constant integer shift = (2 * esize) - UInt(immh:immb);
constant boolean round = FALSE;
```

Assembler Symbols

<**Vb**> Is the destination width specifier, encoded in "immh":

immh	<Vb>
0001	B
001x	H
01xx	S
1xxx	RESERVED

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**Va**> Is the source width specifier, encoded in "immh":

immh	<Va>
0001	H
001x	S
01xx	D
1xxx	RESERVED

<**n**> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<**shift**> For the "Scalar" variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in "immh:immb":

immh	<shift>
0001	16 - UInt(immh:immb)
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	RESERVED

For the "Vector" variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in "immh:immb":

immh	<shift>
0001	16 - UInt(immh:immb)

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

immh	<shift>
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	RESERVED

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

Q	2
0	[absent]
1	[present]

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**Tb**> Is an arrangement specifier, encoded in "immh:Q":

immh	Q	<Tb>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<**Ta**> Is an arrangement specifier, encoded in "immh":

immh	<Ta>
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize*2) operand = V[n, datasize*2];
bits(datasize) result;
integer element;
boolean sat;

for e = 0 to elements-1
    element = RShr(SInt(Elem[operand, e, 2*esize]), shift, round);
    (Elem[result, e, esize], sat) = UnsignedSatQ(element, esize);
    if sat then FPSR.QC = '1';

Vpart[d, part, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.291 SQSUB

Signed saturating subtract

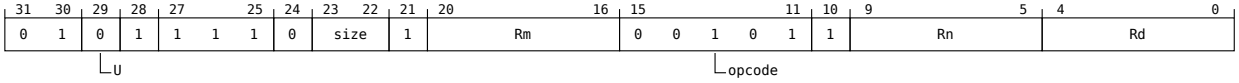
This instruction subtracts the element values of the second source SIMD&FP register from the corresponding element values of the first source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



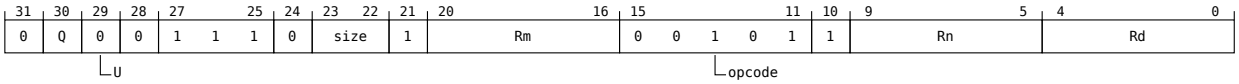
Encoding

```
SQSUB <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
constant boolean unsigned = FALSE;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
SQSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
constant boolean unsigned = FALSE;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<**V**> Is a width specifier, encoded in "size":

size	<V>
00	B
01	H
10	S
11	D

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**n**> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<**m**> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer diff;
boolean sat;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    diff = element1 - element2;
    (Elem[result, e, esize], sat) = SatQ(diff, esize, unsigned);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.292 SQXTN, SQXTN2

Signed saturating extract narrow

This instruction reads each vector element from the source SIMD&FP register, saturates the value to half the original width, places the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements. All the values in this instruction are signed integer values.

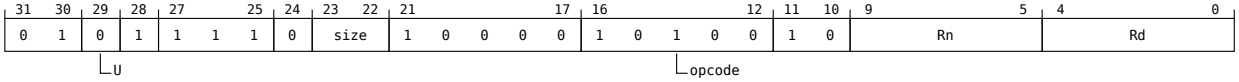
If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The `SQXTN` instruction writes the vector to the lower half of the destination register and clears the upper half. The `SQXTN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



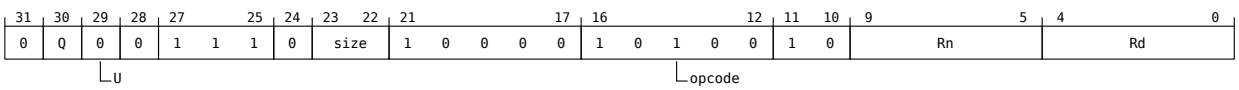
Encoding

`SQXTN <Vb><d>, <Va><n>`

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer part = 0;
constant integer elements = 1;
constant boolean unsigned = FALSE;
```

Vector
(FEAT_AdvSIMD)



Encoding

`SQXTN{2} <Vd>.<Tb>, <Vn>.<Ta>`

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;
constant boolean unsigned = FALSE;
```

Assembler Symbols

<**Vb**> Is the destination width specifier, encoded in "size":

size	<Vb>
00	B
01	H
10	S
11	RESERVED

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**Va**> Is the source width specifier, encoded in "size":

size	<Va>
00	H
01	S
10	D
11	RESERVED

<**n**> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

Q	2
0	[absent]
1	[present]

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**Tb**> Is an arrangement specifier, encoded in "size:Q":

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	Q	<Tb>
10	1	4S
11	x	RESERVED

<**Vn**> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<**Ta**> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(2*datasize) operand = V[n, 2*datasize];
bits(datasize) result;
bits(2*esize) element;
boolean sat;

for e = 0 to elements-1
    element = Elem[operand, e, 2*esize];
    (Elem[result, e, esize], sat) = SatQ(SInt(element), esize, unsigned);
    if sat then FPSR.QC = '1';

Vpart[d, part, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.293 SQXTUN, SQXTUN2

Signed saturating extract unsigned narrow

This instruction reads each signed integer value in the vector of the source SIMD&FP register, saturates the value to an unsigned integer value that is half the original width, places the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements.

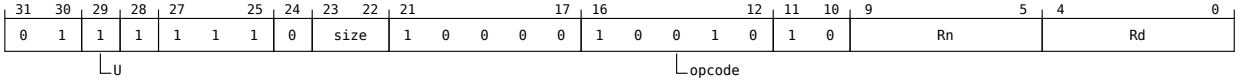
If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The `SQXTUN` instruction writes the vector to the lower half of the destination register and clears the upper half. The `SQXTUN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



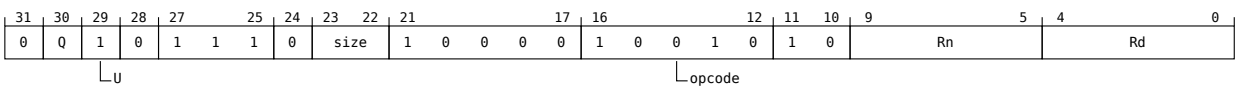
Encoding

```
SQXTUN <Vb><d>, <Va><n>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer part = 0;
constant integer elements = 1;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
SQXTUN{2} <Vd>.<Tb>, <Vn>.<Ta>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer n = UInt(Rn);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Vb**> Is the destination width specifier, encoded in "size":

size	<Vb>
00	B
01	H
10	S
11	RESERVED

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**Va**> Is the source width specifier, encoded in "size":

size	<Va>
00	H
01	S
10	D
11	RESERVED

<**n**> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

Q	2
0	[absent]
1	[present]

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**Tb**> Is an arrangement specifier, encoded in "size:Q":

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	Q	<Tb>
11	x	RESERVED

<**Vn**> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<**Ta**> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(2*datasize) operand = V[n, 2*datasize];
bits(datasize) result;
bits(2*esize) element;
boolean sat;

for e = 0 to elements-1
    element = Elem[operand, e, 2*esize];
    (Elem[result, e, esize], sat) = UnsignedSatQ(SInt(element), esize);
    if sat then FPSR.QC = '1';

Vpart[d, part, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.294 SRHADD

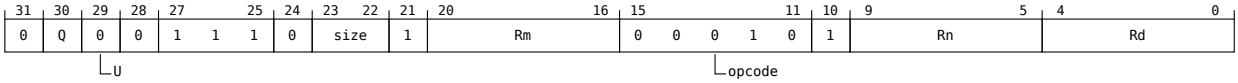
Signed rounding halving add

This instruction adds corresponding signed integer values from the two source SIMD&FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD&FP register.

The results are rounded. For truncated results, see SHADD.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type
(FEAT_AdvSIMD)



Encoding

```
SRHADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    sum = (element1 + element2 + 1) >> 1;
    Elem[result, e, esize] = sum<esize-1:0>;

V[d, datasize] = result;
```

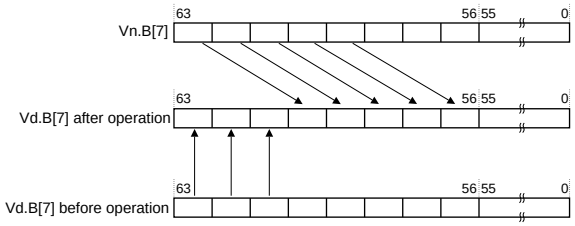

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.295 SRI

Shift right and insert (immediate)

This instruction reads each vector element in the source SIMD&FP register, right shifts each vector element by an immediate value, and inserts the result into the corresponding vector element in the destination SIMD&FP register such that the new zero bits created by the shift are not inserted but retain their existing value. Bits shifted out of the right of each vector element of the source register are lost.

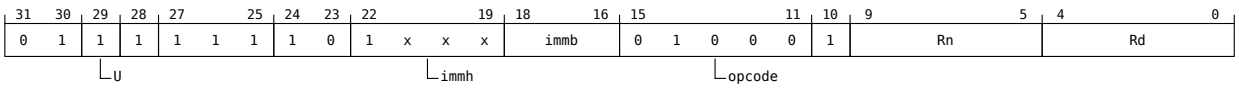
The following figure shows an example of the operation of shift right by 3 for an 8-bit vector element.



Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



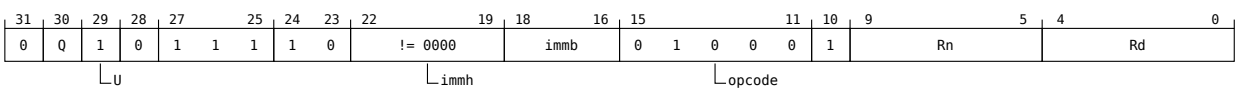
Encoding

```
SRI D<d>, D<n>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh<3> != '1' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << 3;
constant integer datasize = esize;
constant integer elements = 1;
constant integer shift = (esize * 2) - UInt(immh:immb);
```

Vector
(FEAT_AdvSIMD)



Encoding

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
SRI <Vd>.<T>, <Vn>.<T>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh<3>:Q == '10' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << HighestSetBitNZ(immh);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
constant integer shift = (esize * 2) - UInt(immh:immb);
```

Assembler Symbols

<**d**> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<**n**> Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<**shift**> For the “Scalar” variant: is the right shift amount, in the range 1 to 64, encoded as 128 - UInt(“immh:immb”).

For the “Vector” variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

immh	<shift>
0001	16 - UInt(immh:immb)
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	128 - UInt(immh:immb)

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand  = V[n, datasize];
constant bits(datasize) operand2 = V[d, datasize];
constant bits(esize) mask = LSR(Ones(esize), shift);
bits(datasize) result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
bits(esize) shifted;

for e = 0 to elements-1
    shifted = LSR(Elem[operand, e, esize], shift);
    Elem[result, e, esize] = (Elem[operand2, e, esize] AND NOT(mask)) OR shifted;
V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.296 SRSHL

Signed rounding shift left (register)

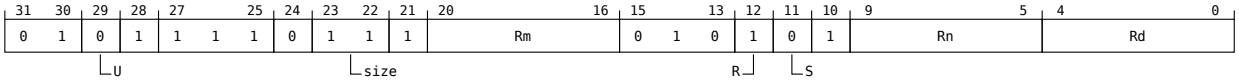
This instruction takes each signed integer value in the vector of the first source SIMD&FP register, shifts it by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a rounding right shift. For a truncating shift, see SSHL.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



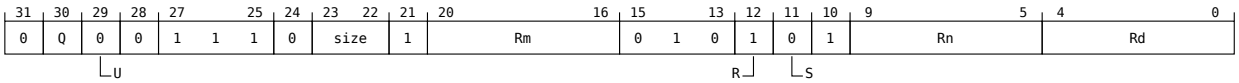
Encoding

```
SRSHL D<d>, D<n>, D<m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if S == '0' && size != '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant boolean rounding = TRUE;
constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
SRSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant boolean rounding = TRUE;
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**d**> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<**n**> Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<**m**> Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

for e = 0 to elements-1
    integer element = SInt(Elem[operand1, e, esize]);
    integer shift = ShiftSat(SInt(Elem[operand2, e, esize]<7:0>), esize);
    if shift >= 0 then // left shift
        element = element << shift;
    else               // right shift
        shift = -shift;
        element = RShr(element, shift, rounding);

    Elem[result, e, esize] = element<esize-1:0>;

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.297 SRSHR

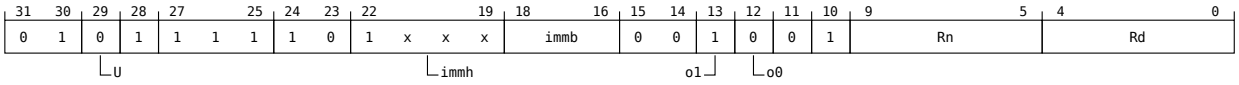
Signed rounding shift right (immediate)

This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, places the final result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values. The results are rounded. For truncated results, see SSHR.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



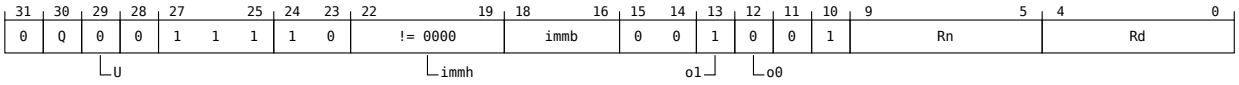
Encoding

```
SRSHR D<d>, D<n>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh<3> != '1' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << 3;
constant integer datasize = esize;
constant integer elements = 1;

constant integer shift = (esize * 2) - UInt(immh:immb);
constant boolean unsigned = FALSE;
constant boolean round = TRUE;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
SRSHR <Vd>.<T>, <Vn>.<T>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh<3>:Q == '10' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << HighestSetBitNZ(immh);
constant integer datasize = 64 << UInt(Q);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer elements = datasize DIV esize;

constant integer shift = (esize * 2) - UInt(immh:immb);
constant boolean unsigned = FALSE;
constant boolean round = TRUE;
```

Assembler Symbols

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**n**> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<**shift**> For the "Scalar" variant: is the right shift amount, in the range 1 to 64, encoded as 128 - UInt("immh:immb").

For the "Vector" variant: is the right shift amount, in the range 1 to the element width in bits, encoded in "immh:immb":

immh	<shift>
0001	16 - UInt(immh:immb)
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	128 - UInt(immh:immb)

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> Is an arrangement specifier, encoded in "immh:Q":

immh	Q	<T>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand  = V[n, datasize];
bits(datasize) result;
integer element;

for e = 0 to elements-1
    constant bits(esize) opelt = Elem[operand, e, esize];
    if unsigned then
        element = RShr(UInt(opelt), shift, round);
    else
        element = RShr(SInt(opelt), shift, round);
    Elem[result, e, esize] = element<esize-1:0>;

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.298 SRSRA

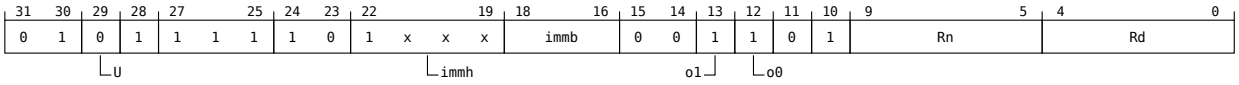
Signed rounding shift right and accumulate (immediate)

This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD&FP register. All the values in this instruction are signed integer values. The results are rounded. For truncated results, see SSRA.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



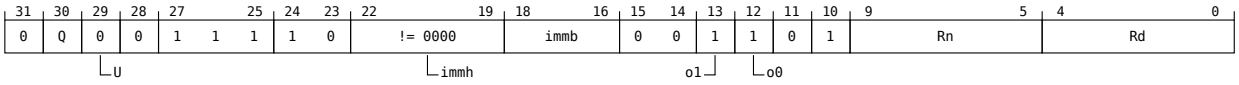
Encoding

```
SRSRA D<d>, D<n>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh<3> != '1' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << 3;
constant integer datasize = esize;
constant integer elements = 1;

constant integer shift = (esize * 2) - UInt(immh:immb);
constant boolean unsigned = FALSE;
constant boolean round = TRUE;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
SRSRA <Vd>.<T>, <Vn>.<T>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh<3>:Q == '10' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << HighestSetBitNZ(immh);
constant integer datasize = 64 << UInt(Q);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer elements = datasize DIV esize;

constant integer shift = (esize * 2) - UInt(immh:immb);
constant boolean unsigned = FALSE;
constant boolean round = TRUE;
```

Assembler Symbols

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**n**> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<**shift**> For the "Scalar" variant: is the right shift amount, in the range 1 to 64, encoded as 128 - UInt("immh:immb").

For the "Vector" variant: is the right shift amount, in the range 1 to the element width in bits, encoded in "immh:immb":

immh	<shift>
0001	16 - UInt(immh:immb)
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	128 - UInt(immh:immb)

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> Is an arrangement specifier, encoded in "immh:Q":

immh	Q	<T>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand  = V[n, datasize];
constant bits(datasize) operand2 = V[d, datasize];
bits(datasize) result;
integer element;

for e = 0 to elements-1
    constant bits(esize) opelt = Elem[operand, e, esize];
    if unsigned then
        element = RShr(UInt(opelt), shift, round);
    else
        element = RShr(SInt(opelt), shift, round);
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.299 SSHL

Signed shift left (register)

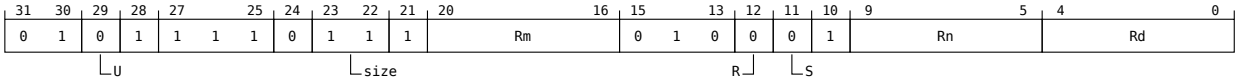
This instruction takes each signed integer value in the vector of the first source SIMD&FP register, shifts each value by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a truncating right shift. For a rounding shift, see SRSHL.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



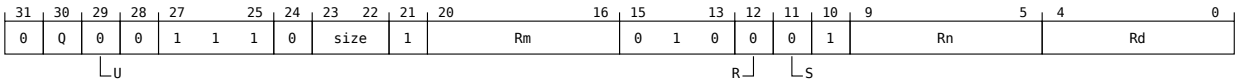
Encoding

```
SSHL D<d>, D<n>, D<m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if S == '0' && size != '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant boolean rounding = FALSE;
constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
SSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant boolean rounding = FALSE;
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**d**> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<**n**> Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<**m**> Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

for e = 0 to elements-1
    integer element = SInt(Elem[operand1, e, esize]);
    integer shift = ShiftSat(SInt(Elem[operand2, e, esize]<7:0>), esize);
    if shift >= 0 then // left shift
        element = element << shift;
    else               // right shift
        shift = -shift;
        element = RShr(element, shift, rounding);

    Elem[result, e, esize] = element<esize-1:0>;

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.300 SSHLL, SSHLL2

Signed shift left long (immediate)

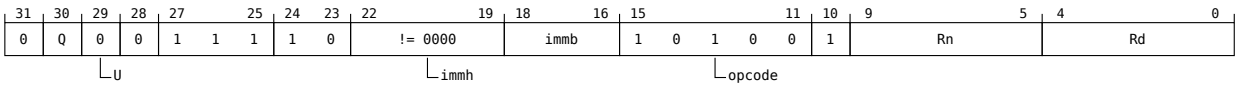
This instruction reads each vector element from the source SIMD&FP register, left shifts each vector element by the specified shift amount, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are signed integer values.

The SSHLL instruction extracts vector elements from the lower half of the source register. The SSHLL2 instruction extracts vector elements from the upper half of the source register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias SXTL, SXTL2.

Vector
(FEAT_AdvSIMD)



Encoding

```
SSHLL{2} <Vd>.<Ta>, <Vn>.<Tb>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh<3> == '1' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << HighestSetBitNZ(immh<2:0>);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;

constant integer shift = UInt(immh:immb) - esize;
constant boolean unsigned = FALSE;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Ta**> Is an arrangement specifier, encoded in “immh”:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

immh	<Ta>
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<**Tb**> Is an arrangement specifier, encoded in "immh:Q":

immh	Q	<Tb>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<**shift**> Is the left shift amount, in the range 0 to the source element width in bits minus 1, encoded in "immh:immb":

immh	<shift>
0001	UInt(immh:immb) - 8
001x	UInt(immh:immb) - 16
01xx	UInt(immh:immb) - 32
1xxx	RESERVED

Alias Conditions

Alias	Is preferred when
SXTL, SXTL2	immb == '000' &&BitCount(immh) == 1

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = Vpart[n, part, datasize];
bits(datasize*2) result;

for e = 0 to elements-1
    constant bits(esize) opelt = Elem[operand, e, esize];
    constant integer element = if unsigned then UInt(opelt) else SInt(opelt);
    Elem[result, e, 2*esize] = (element << shift)<2*esize-1:0>;

V[d, datasize*2] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.301 SSHR

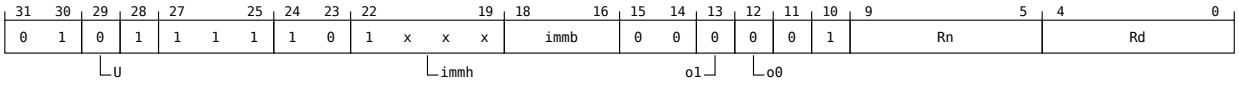
Signed shift right (immediate)

This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, places the final result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values. The results are truncated. For rounded results, see SRSHR.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



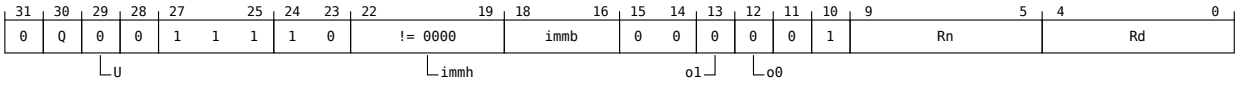
Encoding

```
SSHR D<d>, D<n>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh<3> != '1' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << 3;
constant integer datasize = esize;
constant integer elements = 1;

constant integer shift = (esize * 2) - UInt(immh:immb);
constant boolean unsigned = FALSE;
constant boolean round = FALSE;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
SSHR <Vd>.<T>, <Vn>.<T>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh<3>:Q == '10' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << HighestSetBitNZ(immh);
constant integer datasize = 64 << UInt(Q);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer elements = datasize DIV esize;

constant integer shift = (esize * 2) - UInt(immh:immb);
constant boolean unsigned = FALSE;
constant boolean round = FALSE;
```

Assembler Symbols

<**d**> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<**n**> Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<**shift**> For the “Scalar” variant: is the right shift amount, in the range 1 to 64, encoded as 128 - UInt(“immh:immb”).

For the “Vector” variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

immh	<shift>
0001	16 - UInt(immh:immb)
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	128 - UInt(immh:immb)

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand  = V[n, datasize];
bits(datasize) result;
integer element;

for e = 0 to elements-1
    constant bits(esize) opelt = Elem[operand, e, esize];
    if unsigned then
        element = RShr(UInt(opelt), shift, round);
    else
        element = RShr(SInt(opelt), shift, round);
    Elem[result, e, esize] = element<esize-1:0>;

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.302 SSRA

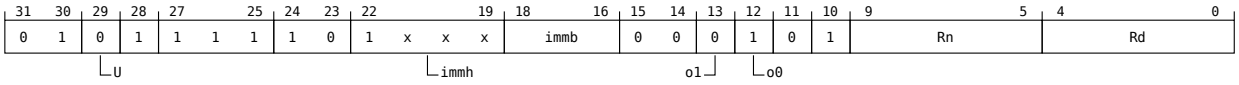
Signed shift right and accumulate (immediate)

This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD&FP register. All the values in this instruction are signed integer values. The results are truncated. For rounded results, see SRSRA.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



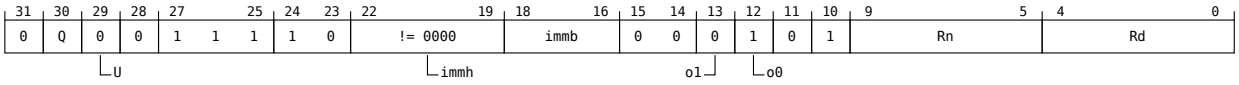
Encoding

```
SSRA D<d>, D<n>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh<3> != '1' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << 3;
constant integer datasize = esize;
constant integer elements = 1;

constant integer shift = (esize * 2) - UInt(immh:immb);
constant boolean unsigned = FALSE;
constant boolean round = FALSE;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
SSRA <Vd>.<T>, <Vn>.<T>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh<3>:Q == '10' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << HighestSetBitNZ(immh);
constant integer datasize = 64 << UInt(Q);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer elements = datasize DIV esize;

constant integer shift = (esize * 2) - UInt(immh:immb);
constant boolean unsigned = FALSE;
constant boolean round = FALSE;
```

Assembler Symbols

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**n**> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<**shift**> For the "Scalar" variant: is the right shift amount, in the range 1 to 64, encoded as 128 - UInt("immh:immb").

For the "Vector" variant: is the right shift amount, in the range 1 to the element width in bits, encoded in "immh:immb":

immh	<shift>
0001	16 - UInt(immh:immb)
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	128 - UInt(immh:immb)

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> Is an arrangement specifier, encoded in "immh:Q":

immh	Q	<T>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand  = V[n, datasize];
constant bits(datasize) operand2 = V[d, datasize];
bits(datasize) result;
integer element;

for e = 0 to elements-1
    constant bits(esize) opelt = Elem[operand, e, esize];
    if unsigned then
        element = RShr(UInt(opelt), shift, round);
    else
        element = RShr(SInt(opelt), shift, round);
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.303 SSUBL, SSUBL2

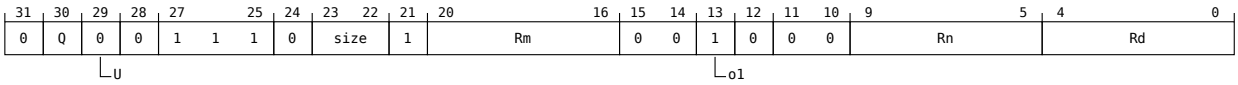
Signed subtract long

This instruction subtracts each vector element in the lower or upper half of the second source SIMD&FP register from the corresponding vector element of the first source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values. The destination vector elements are twice as long as the source vector elements.

The `SSUBL` instruction extracts each source vector from the lower half of each source register. The `SSUBL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers, not all the same type
(FEAT_AdvSIMD)



Encoding

```
SSUBL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "size":

size	<Ta>
00	8H

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<Ta>
01	4S
10	2D
11	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Tb**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = Vpart[n, part, datasize];
constant bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    sum = element1 - element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d, 2*datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.304 SSUBW, SSUBW2

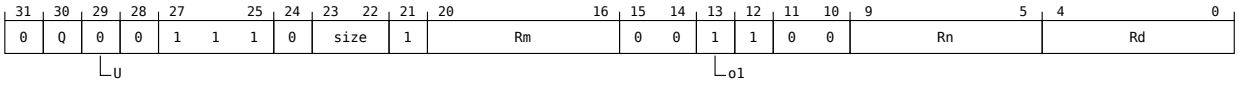
Signed subtract wide

This instruction subtracts each vector element in the lower or upper half of the second source SIMD&FP register from the corresponding vector element in the first source SIMD&FP register, places the result in a vector, and writes the vector to the SIMD&FP destination register. All the values in this instruction are signed integer values.

The `SSUBW` instruction extracts the second source vector from the lower half of the second source register. The `SSUBW2` instruction extracts the second source vector from the upper half of the second source register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers, not all the same type
(FEAT_AdvSIMD)



Encoding

```
SSUBW{2} <Vd>.<Ta>, <Vn>.<Ta>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "size":

size	<Ta>
00	8H
01	4S

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<Ta>
10	2D
11	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

<**Tb**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(2*datasize) operand1 = V[n, 2*datasize];
constant bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, 2*esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    sum = element1 - element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d, 2*datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.305 ST1 (multiple structures)

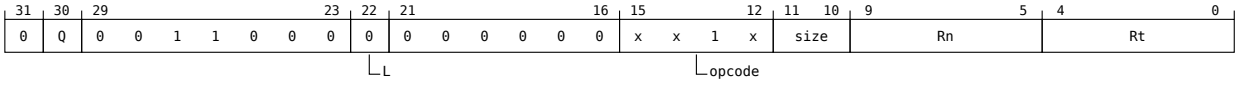
Store multiple single-element structures from one, two, three, or four registers

This instruction stores elements to memory from one, two, three, or four SIMD&FP registers, without interleaving. Every element of each register is stored.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset
(FEAT_AdvSIMD)



Encoding for the One register variant Applies when `(opcode == 0111)`

```
ST1 { <Vt>.<T> }, [<Xn|SP>]
```

Encoding for the Two registers variant Applies when `(opcode == 1010)`

```
ST1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]
```

Encoding for the Three registers variant Applies when `(opcode == 0110)`

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]
```

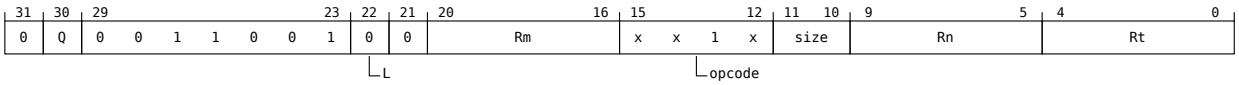
Encoding for the Four registers variant Applies when `(opcode == 0010)`

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = integer UNKNOWN;
constant boolean wback = FALSE;
constant boolean nontemporal = FALSE;
```

Post-index
(FEAT_AdvSIMD)



Encoding for the One register, immediate offset variant Applies when `(Rm == 11111 && opcode == 0111)`

```
ST1 { <Vt>.<T> }, [<Xn|SP>], <imm>
```

Encoding for the One register, register offset variant Applies when `(Rm != 11111 && opcode == 0111)`

```
ST1 { <Vt>.<T> }, [<Xn|SP>], <Xm>
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the Two registers, immediate offset variant Applies when `(Rm == 11111 && opcode == 1010)`

```
ST1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>
```

Encoding for the Two registers, register offset variant Applies when `(Rm != 11111 && opcode == 1010)`

```
ST1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>
```

Encoding for the Three registers, immediate offset variant Applies when `(Rm == 11111 && opcode == 0110)`

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>
```

Encoding for the Three registers, register offset variant Applies when `(Rm != 11111 && opcode == 0110)`

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>
```

Encoding for the Four registers, immediate offset variant Applies when `(Rm == 11111 && opcode == 0010)`

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>
```

Encoding for the Four registers, register offset variant Applies when `(Rm != 11111 && opcode == 0010)`

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant boolean wback = TRUE;
constant boolean nontemporal = FALSE;
```

Assembler Symbols

<**Vt**> Is the name of the first or only SIMD&FP register to be transferred, encoded in the “Rt” field.

<**T**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	1D
11	1	2D

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<**Vt2**> Is the name of the second SIMD&FP register to be transferred, encoded as “Rt” plus 1 modulo 32.

<**Vt3**> Is the name of the third SIMD&FP register to be transferred, encoded as “Rt” plus 2 modulo 32.

<**Vt4**> Is the name of the fourth SIMD&FP register to be transferred, encoded as “Rt” plus 3 modulo 32.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**imm**> For the “One register, immediate offset” variant: is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#8
1	#16

For the “Two registers, immediate offset” variant: is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#16
1	#32

For the “Three registers, immediate offset” variant: is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#24
1	#48

For the “Four registers, immediate offset” variant: is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#32
1	#64

<**Xm**> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

Shared Decode

```
constant integer datasize = 64 << UInt(Q);
constant integer esize = 8 << UInt(size);
constant integer elements = datasize DIV esize;

integer rpt;               // number of iterations
constant integer selem = 1; // structure elements

case opcode of
    when '0010' rpt = 4;    // LD/ST1 (4 registers)
    when '0110' rpt = 3;    // LD/ST1 (3 registers)
    when '1010' rpt = 2;    // LD/ST1 (2 registers)
    when '0111' rpt = 1;    // LD/ST1 (1 register)
    otherwise EndOfDecode(Decode_UNDEF);
```

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(64) address;
bits(64) eaddr;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_STORE, nontemporal, privileged);
if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

offs = Zeros(64);
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt, datasize];
            eaddr = AddressIncrement(address, offs, accdesc);
            Mem[eaddr, ebytes, accdesc] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;
if wback then
    if m != 31 then
        offs = X[m, 64];
    address = AddressAdd(address, offs, accdesc);
    if n == 31 then
        SP[64] = address;
    else
        X[n, 64] = address;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.306 ST1 (single structure)

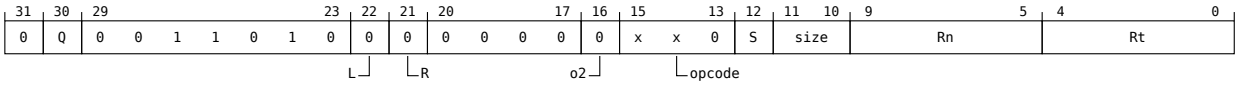
Store a single-element structure from one lane of one register

This instruction stores the specified element of a SIMD&FP register to memory.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset
(FEAT_AdvSIMD)



Encoding for the 8-bit variant Applies when `(opcode == 000)`

```
ST1 { <Vt>.B }[<index>], [<Xn|SP>]
```

Encoding for the 16-bit variant Applies when `(opcode == 010 && size == x0)`

```
ST1 { <Vt>.H }[<index>], [<Xn|SP>]
```

Encoding for the 32-bit variant Applies when `(opcode == 100 && size == 00)`

```
ST1 { <Vt>.S }[<index>], [<Xn|SP>]
```

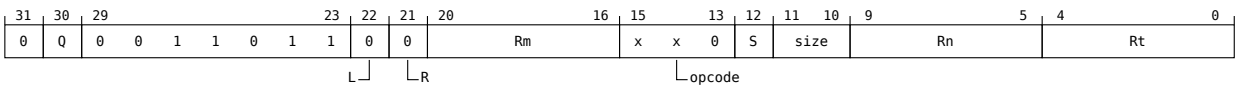
Encoding for the 64-bit variant Applies when `(opcode == 100 && S == 0 && size == 01)`

```
ST1 { <Vt>.D }[<index>], [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = integer UNKNOWN;
constant boolean wback = FALSE;
constant boolean nontemporal = FALSE;
```

Post-index
(FEAT_AdvSIMD)



Encoding for the 8-bit, immediate offset variant Applies when `(Rm == 11111 && opcode == 000)`

```
ST1 { <Vt>.B }[<index>], [<Xn|SP>], #1
```

Encoding for the 8-bit, register offset variant Applies when `(Rm != 11111 && opcode == 000)`

```
ST1 { <Vt>.B }[<index>], [<Xn|SP>], <Xm>
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 16-bit, immediate offset variant Applies when `(Rm == 11111 && opcode == 010 && size == x0)`

```
ST1 { <Vt>.H }[<index>], [<Xn|SP>], #2
```

Encoding for the 16-bit, register offset variant Applies when `(Rm != 11111 && opcode == 010 && size == x0)`

```
ST1 { <Vt>.H }[<index>], [<Xn|SP>], <Xm>
```

Encoding for the 32-bit, immediate offset variant Applies when `(Rm == 11111 && opcode == 100 && size == 00)`

```
ST1 { <Vt>.S }[<index>], [<Xn|SP>], #4
```

Encoding for the 32-bit, register offset variant Applies when `(Rm != 11111 && opcode == 100 && size == 00)`

```
ST1 { <Vt>.S }[<index>], [<Xn|SP>], <Xm>
```

Encoding for the 64-bit, immediate offset variant Applies when `(Rm == 11111 && opcode == 100 && S == 0 && size == 01)`

```
ST1 { <Vt>.D }[<index>], [<Xn|SP>], #8
```

Encoding for the 64-bit, register offset variant Applies when `(Rm != 11111 && opcode == 100 && S == 0 && size == 01)`

```
ST1 { <Vt>.D }[<index>], [<Xn|SP>], <Xm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant boolean wback = TRUE;
constant boolean nontemporal = FALSE;
```

Assembler Symbols

<**Vt**> Is the name of the first or only SIMD&FP register to be transferred, encoded in the “Rt” field.

<**index**> For the “8-bit”, “8-bit, immediate offset”, and “8-bit, register offset” variants: is the element index, encoded in “Q:S:size”.

For the “16-bit”, “16-bit, immediate offset”, and “16-bit, register offset” variants: is the element index, encoded in “Q:S:size<1>”.

For the “32-bit”, “32-bit, immediate offset”, and “32-bit, register offset” variants: is the element index, encoded in “Q:S”.

For the “64-bit”, “64-bit, immediate offset”, and “64-bit, register offset” variants: is the element index, encoded in “Q”.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<**Xm**> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

Shared Decode

```
bits(2) scale = opcode<2:1>;
constant integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when '11'
        // load and replicate
        if L == '0' || S == '1' then EndOfDecode(Decode_UNDEF);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        scale = size;
        replicate = TRUE;
    when '00'
        index = UInt(Q:S:size);       // B[0-15]
    when '01'
        if size<0> == '1' then EndOfDecode(Decode_UNDEF);
        index = UInt(Q:S:size<1>);    // H[0-7]
    when '10'
        if size<1> == '1' then EndOfDecode(Decode_UNDEF);
        if size<0> == '0' then
            index = UInt(Q:S);        // S[0-3]
        else
            if S == '1' then EndOfDecode(Decode_UNDEF);
            index = UInt(Q);          // D[0-1]
            scale = '11';

constant integer datasize = 64 << UInt(Q);
constant integer esize = 8 << UInt(scale);
```

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(64) address;
bits(64) eaddr;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_STORE, nontemporal, privileged);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

offs = Zeros(64);

if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        eaddr = AddressIncrement(address, offs, accdesc);
        element = Mem[eaddr, ebytes, accdesc];
        // replicate to fill 128- or 64-bit register
        V[t, datasize] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t, 128];
        eaddr = AddressIncrement(address, offs, accdesc);
        // extract from one lane of 128-bit register
        Mem[eaddr, ebytes, accdesc] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = ( t + 1 ) MOD 32;
if wback then
    if m != 31 then
        offs = X[m, 64];
    address = AddressAdd(address, offs, accdesc);
    if n == 31 then
        SP[64] = address;
    else
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        X[n, 64] = address;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.307 ST2 (multiple structures)

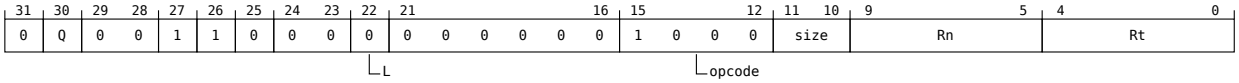
Store multiple 2-element structures from two registers

This instruction stores multiple 2-element structures from two SIMD&FP registers to memory, with interleaving. Every element of each register is stored.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset
(FEAT_AdvSIMD)



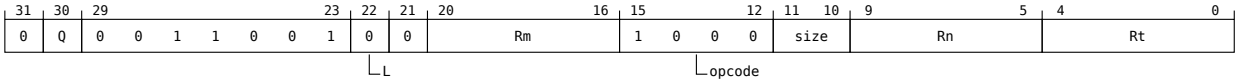
Encoding

```
ST2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = integer UNKNOWN;
constant boolean wback = FALSE;
constant boolean nontemporal = FALSE;
```

Post-index
(FEAT_AdvSIMD)



Encoding for the Immediate offset variant Applies when `(Rm == 11111)`

```
ST2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>
```

Encoding for the Register offset variant Applies when `(Rm != 11111)`

```
ST2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant boolean wback = TRUE;
constant boolean nontemporal = FALSE;
```

Assembler Symbols

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**Vt**> Is the name of the first or only SIMD&FP register to be transferred, encoded in the “Rt” field.

<**T**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<**Vt2**> Is the name of the second SIMD&FP register to be transferred, encoded as “Rt” plus 1 modulo 32.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<**imm**> Is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#16
1	#32

<**Xm**> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

Shared Decode

```
constant integer datasize = 64 << UInt(Q);
constant integer esize = 8 << UInt(size);
constant integer elements = datasize DIV esize;

constant integer rpt = 1;
constant integer selem = 2;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then EndOfDecode(Decode_UNDEF);
```

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(64) address;
bits(64) eaddr;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_STORE, nontemporal, privileged);
if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
offs = Zeros(64);
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt, datasize];
            eaddr = AddressIncrement(address, offs, accdesc);
            Mem[eaddr, ebytes, accdesc] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;
if wback then
    if m != 31 then
        offs = X[m, 64];
    address = AddressAdd(address, offs, accdesc);
    if n == 31 then
        SP[64] = address;
    else
        X[n, 64] = address;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.308 ST2 (single structure)

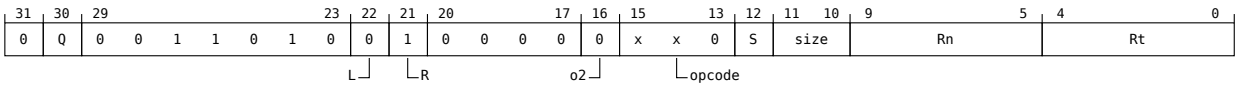
Store single 2-element structure from one lane of two registers

This instruction stores a 2-element structure to memory from corresponding elements of two SIMD&FP registers.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset
(FEAT_AdvSIMD)



Encoding for the 8-bit variant Applies when `(opcode == 000)`

```
ST2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>]
```

Encoding for the 16-bit variant Applies when `(opcode == 010 && size == x0)`

```
ST2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>]
```

Encoding for the 32-bit variant Applies when `(opcode == 100 && size == 00)`

```
ST2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>]
```

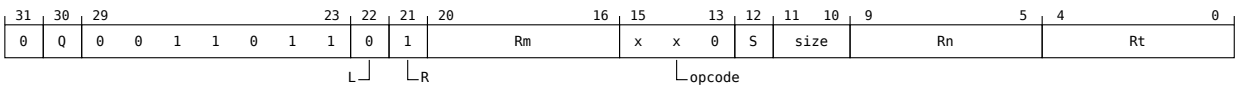
Encoding for the 64-bit variant Applies when `(opcode == 100 && S == 0 && size == 01)`

```
ST2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = integer UNKNOWN;
constant boolean wback = FALSE;
constant boolean nontemporal = FALSE;
```

Post-index
(FEAT_AdvSIMD)



Encoding for the 8-bit, immediate offset variant Applies when `(Rm == 11111 && opcode == 000)`

```
ST2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], #2
```

Encoding for the 8-bit, register offset variant Applies when `(Rm != 11111 && opcode == 000)`

```
ST2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], <Xm>
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 16-bit, immediate offset variant Applies when `(Rm == 11111 && opcode == 010 && size == x0)`

```
ST2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], #4
```

Encoding for the 16-bit, register offset variant Applies when `(Rm != 11111 && opcode == 010 && size == x0)`

```
ST2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], <Xm>
```

Encoding for the 32-bit, immediate offset variant Applies when `(Rm == 11111 && opcode == 100 && size == 00)`

```
ST2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>], #8
```

Encoding for the 32-bit, register offset variant Applies when `(Rm != 11111 && opcode == 100 && size == 00)`

```
ST2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>], <Xm>
```

Encoding for the 64-bit, immediate offset variant Applies when `(Rm == 11111 && opcode == 100 && S == 0 && size == 01)`

```
ST2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], #16
```

Encoding for the 64-bit, register offset variant Applies when `(Rm != 11111 && opcode == 100 && S == 0 && size == 01)`

```
ST2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], <Xm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant boolean wback = TRUE;
constant boolean nontemporal = FALSE;
```

Assembler Symbols

<**Vt**> Is the name of the first or only SIMD&FP register to be transferred, encoded in the “Rt” field.

<**Vt2**> Is the name of the second SIMD&FP register to be transferred, encoded as “Rt” plus 1 modulo 32.

<**index**> For the “8-bit”, “8-bit, immediate offset”, and “8-bit, register offset” variants: is the element index, encoded in “Q:S:size”.

For the “16-bit”, “16-bit, immediate offset”, and “16-bit, register offset” variants: is the element index, encoded in “Q:S:size<1>”.

For the “32-bit”, “32-bit, immediate offset”, and “32-bit, register offset” variants: is the element index, encoded in “Q:S”.

For the “64-bit”, “64-bit, immediate offset”, and “64-bit, register offset” variants: is the element index, encoded in “Q”.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<**Xm**> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

Shared Decode

```
bits(2) scale = opcode<2:1>;
constant integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when '11'
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        // load and replicate
        if L == '0' || S == '1' then EndOfDecode(Decode_UNDEF);
        scale = size;
        replicate = TRUE;
    when '00'
        index = UInt(Q:S:size);       // B[0-15]
    when '01'
        if size<0> == '1' then EndOfDecode(Decode_UNDEF);
        index = UInt(Q:S:size<1>);    // H[0-7]
    when '10'
        if size<1> == '1' then EndOfDecode(Decode_UNDEF);
        if size<0> == '0' then
            index = UInt(Q:S);        // S[0-3]
        else
            if S == '1' then EndOfDecode(Decode_UNDEF);
            index = UInt(Q);          // D[0-1]
            scale = '11';

constant integer datasize = 64 << UInt(Q);
constant integer esize = 8 << UInt(scale);
```

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(64) address;
bits(64) eaddr;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_STORE, nontemporal, privileged);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

offs = Zeros(64);

if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        eaddr = AddressIncrement(address, offs, accdesc);
        element = Mem[eaddr, ebytes, accdesc];
        // replicate to fill 128- or 64-bit register
        V[t, datasize] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t, 128];
        eaddr = AddressIncrement(address, offs, accdesc);
        // extract from one lane of 128-bit register
        Mem[eaddr, ebytes, accdesc] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = ( t + 1 ) MOD 32;
if wback then
    if m != 31 then
        offs = X[m, 64];
    address = AddressAdd(address, offs, accdesc);
    if n == 31 then
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        SP[64] = address;
    else
        X[n, 64] = address;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.309 ST3 (multiple structures)

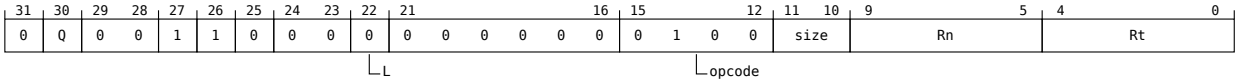
Store multiple 3-element structures from three registers

This instruction stores multiple 3-element structures to memory from three SIMD&FP registers, with interleaving. Every element of each register is stored.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset
(FEAT_AdvSIMD)



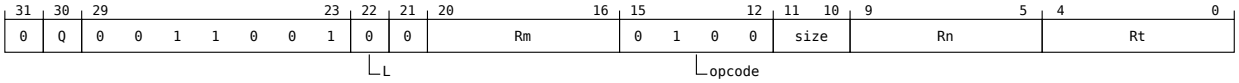
Encoding

```
ST3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = integer UNKNOWN;
constant boolean wback = FALSE;
constant boolean nontemporal = FALSE;
```

Post-index
(FEAT_AdvSIMD)



Encoding for the Immediate offset variant Applies when `(Rm == 11111)`

```
ST3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>
```

Encoding for the Register offset variant Applies when `(Rm != 11111)`

```
ST3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant boolean wback = TRUE;
constant boolean nontemporal = FALSE;
```

Assembler Symbols

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**Vt**> Is the name of the first or only SIMD&FP register to be transferred, encoded in the “Rt” field.

<**T**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<**Vt2**> Is the name of the second SIMD&FP register to be transferred, encoded as “Rt” plus 1 modulo 32.

<**Vt3**> Is the name of the third SIMD&FP register to be transferred, encoded as “Rt” plus 2 modulo 32.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<**imm**> Is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#24
1	#48

<**Xm**> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

Shared Decode

```
constant integer datasize = 64 << UInt(Q);
constant integer esize = 8 << UInt(size);
constant integer elements = datasize DIV esize;

constant integer rpt = 1;
constant integer selem = 3;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then EndOfDecode(Decode_UNDEF);
```

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(64) address;
bits(64) eaddr;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_STORE, nontemporal, privileged);
if n == 31 then
    CheckSPAlignment();
    address = SP[64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    address = X[n, 64];

offs = Zeros(64);
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt, datasize];
            eaddr = AddressIncrement(address, offs, accdesc);
            Mem[eaddr, ebytes, accdesc] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;
if wback then
    if m != 31 then
        offs = X[m, 64];
    address = AddressAdd(address, offs, accdesc);
    if n == 31 then
        SP[64] = address;
    else
        X[n, 64] = address;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.310 ST3 (single structure)

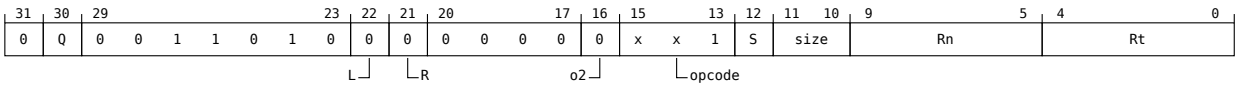
Store single 3-element structure from one lane of three registers

This instruction stores a 3-element structure to memory from corresponding elements of three SIMD&FP registers.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset
(FEAT_AdvSIMD)



Encoding for the 8-bit variant Applies when `(opcode == 001)`

```
ST3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>]
```

Encoding for the 16-bit variant Applies when `(opcode == 011 && size == x0)`

```
ST3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>]
```

Encoding for the 32-bit variant Applies when `(opcode == 101 && size == 00)`

```
ST3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>]
```

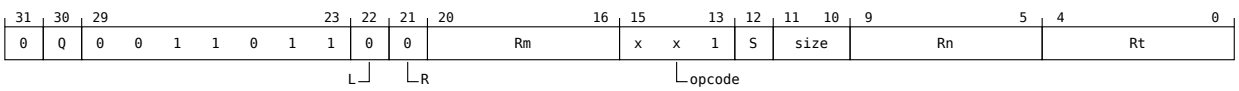
Encoding for the 64-bit variant Applies when `(opcode == 101 && S == 0 && size == 01)`

```
ST3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = integer UNKNOWN;
constant boolean wback = FALSE;
constant boolean nontemporal = FALSE;
```

Post-index
(FEAT_AdvSIMD)



Encoding for the 8-bit, immediate offset variant Applies when `(Rm == 11111 && opcode == 001)`

```
ST3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], #3
```

Encoding for the 8-bit, register offset variant Applies when `(Rm != 11111 && opcode == 001)`

```
ST3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], <Xm>
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 16-bit, immediate offset variant Applies when `(Rm == 11111 && opcode == 011 && size == x0)`

```
ST3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], #6
```

Encoding for the 16-bit, register offset variant Applies when `(Rm != 11111 && opcode == 011 && size == x0)`

```
ST3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], <Xm>
```

Encoding for the 32-bit, immediate offset variant Applies when `(Rm == 11111 && opcode == 101 && size == 00)`

```
ST3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], #12
```

Encoding for the 32-bit, register offset variant Applies when `(Rm != 11111 && opcode == 101 && size == 00)`

```
ST3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], <Xm>
```

Encoding for the 64-bit, immediate offset variant Applies when `(Rm == 11111 && opcode == 101 && S == 0 && size == 01)`

```
ST3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], #24
```

Encoding for the 64-bit, register offset variant Applies when `(Rm != 11111 && opcode == 101 && S == 0 && size == 01)`

```
ST3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], <Xm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant boolean wback = TRUE;
constant boolean nontemporal = FALSE;
```

Assembler Symbols

<**Vt**> Is the name of the first or only SIMD&FP register to be transferred, encoded in the “Rt” field.

<**Vt2**> Is the name of the second SIMD&FP register to be transferred, encoded as “Rt” plus 1 modulo 32.

<**Vt3**> Is the name of the third SIMD&FP register to be transferred, encoded as “Rt” plus 2 modulo 32.

<**index**> For the “8-bit”, “8-bit, immediate offset”, and “8-bit, register offset” variants: is the element index, encoded in “Q:S:size”.

For the “16-bit”, “16-bit, immediate offset”, and “16-bit, register offset” variants: is the element index, encoded in “Q:S:size<1>”.

For the “32-bit”, “32-bit, immediate offset”, and “32-bit, register offset” variants: is the element index, encoded in “Q:S”.

For the “64-bit”, “64-bit, immediate offset”, and “64-bit, register offset” variants: is the element index, encoded in “Q”.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<**Xm**> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

Shared Decode

```
bits(2) scale = opcode<2:1>;
constant integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
case scale of
    when '11'
        // load and replicate
        if L == '0' || S == '1' then EndOfDecode(Decode_UNDEF);
        scale = size;
        replicate = TRUE;
    when '00'
        index = UInt(Q:S:size);       // B[0-15]
    when '01'
        if size<0> == '1' then EndOfDecode(Decode_UNDEF);
        index = UInt(Q:S:size<1>);    // H[0-7]
    when '10'
        if size<1> == '1' then EndOfDecode(Decode_UNDEF);
        if size<0> == '0' then
            index = UInt(Q:S);        // S[0-3]
        else
            if S == '1' then EndOfDecode(Decode_UNDEF);
            index = UInt(Q);          // D[0-1]
            scale = '11';

constant integer datasize = 64 << UInt(Q);
constant integer esize = 8 << UInt(scale);
```

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(64) address;
bits(64) eaddr;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_STORE, nontemporal, privileged);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

offs = Zeros(64);

if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        eaddr = AddressIncrement(address, offs, accdesc);
        element = Mem[eaddr, ebytes, accdesc];
        // replicate to fill 128- or 64-bit register
        V[t, datasize] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t, 128];
        eaddr = AddressIncrement(address, offs, accdesc);
        // extract from one lane of 128-bit register
        Mem[eaddr, ebytes, accdesc] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = ( t + 1 ) MOD 32;
if wback then
    if m != 31 then
        offs = X[m, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
address = AddressAdd(address, offs, accdesc);
if n == 31 then
    SP[64] = address;
else
    X[n, 64] = address;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.311 ST4 (multiple structures)

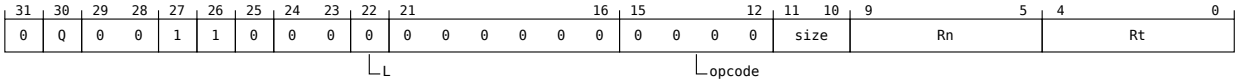
Store multiple 4-element structures from four registers

This instruction stores multiple 4-element structures to memory from four SIMD&FP registers, with interleaving. Every element of each register is stored.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset
(FEAT_AdvSIMD)



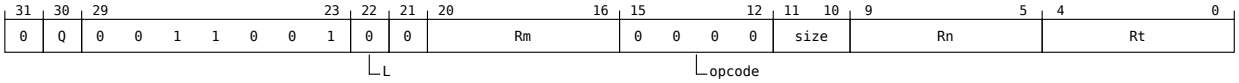
Encoding

```
ST4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = integer UNKNOWN;
constant boolean wback = FALSE;
constant boolean nontemporal = FALSE;
```

Post-index
(FEAT_AdvSIMD)



Encoding for the Immediate offset variant Applies when `(Rm == 11111)`

```
ST4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>
```

Encoding for the Register offset variant Applies when `(Rm != 11111)`

```
ST4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant boolean wback = TRUE;
constant boolean nontemporal = FALSE;
```

Assembler Symbols

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**Vt**> Is the name of the first or only SIMD&FP register to be transferred, encoded in the “Rt” field.

<**T**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<**Vt2**> Is the name of the second SIMD&FP register to be transferred, encoded as “Rt” plus 1 modulo 32.

<**Vt3**> Is the name of the third SIMD&FP register to be transferred, encoded as “Rt” plus 2 modulo 32.

<**Vt4**> Is the name of the fourth SIMD&FP register to be transferred, encoded as “Rt” plus 3 modulo 32.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<**imm**> Is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#32
1	#64

<**Xm**> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

Shared Decode

```
constant integer datasize = 64 << UInt(Q);
constant integer esize = 8 << UInt(size);
constant integer elements = datasize DIV esize;

constant integer rpt = 1;
constant integer selem = 4;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then EndOfDecode(Decode_UNDEF);
```

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(64) address;
bits(64) eaddr;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_STORE, nontemporal, privileged);
if n == 31 then
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

offs = Zeros(64);
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt, datasize];
            eaddr = AddressIncrement(address, offs, accdesc);
            Mem[eaddr, ebytes, accdesc] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;
if wback then
    if m != 31 then
        offs = X[m, 64];
    address = AddressAdd(address, offs, accdesc);
    if n == 31 then
        SP[64] = address;
    else
        X[n, 64] = address;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.312 ST4 (single structure)

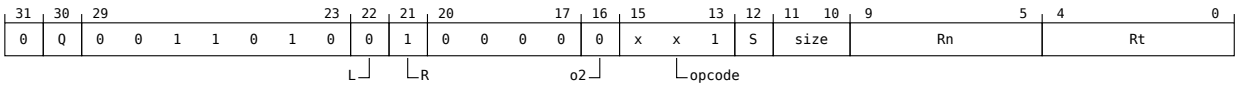
Store single 4-element structure from one lane of four registers

This instruction stores a 4-element structure to memory from corresponding elements of four SIMD&FP registers.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset
(FEAT_AdvSIMD)



Encoding for the 8-bit variant Applies when `(opcode == 001)`

```
ST4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>]
```

Encoding for the 16-bit variant Applies when `(opcode == 011 && size == x0)`

```
ST4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>]
```

Encoding for the 32-bit variant Applies when `(opcode == 101 && size == 00)`

```
ST4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>]
```

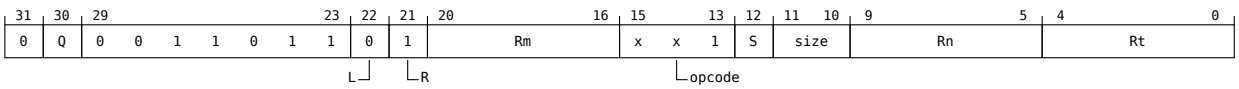
Encoding for the 64-bit variant Applies when `(opcode == 101 && S == 0 && size == 01)`

```
ST4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = integer UNKNOWN;
constant boolean wback = FALSE;
constant boolean nontemporal = FALSE;
```

Post-index
(FEAT_AdvSIMD)



Encoding for the 8-bit, immediate offset variant Applies when `(Rm == 11111 && opcode == 001)`

```
ST4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>], #4
```

Encoding for the 8-bit, register offset variant Applies when `(Rm != 11111 && opcode == 001)`

```
ST4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>], <Xm>
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 16-bit, immediate offset variant Applies when `(Rm == 11111 && opcode == 011 && size == x0)`

```
ST4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>], #8
```

Encoding for the 16-bit, register offset variant Applies when `(Rm != 11111 && opcode == 011 && size == x0)`

```
ST4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>], <Xm>
```

Encoding for the 32-bit, immediate offset variant Applies when `(Rm == 11111 && opcode == 101 && size == 00)`

```
ST4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>], #16
```

Encoding for the 32-bit, register offset variant Applies when `(Rm != 11111 && opcode == 101 && size == 00)`

```
ST4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>], <Xm>
```

Encoding for the 64-bit, immediate offset variant Applies when `(Rm == 11111 && opcode == 101 && S == 0 && size == 01)`

```
ST4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>], #32
```

Encoding for the 64-bit, register offset variant Applies when `(Rm != 11111 && opcode == 101 && S == 0 && size == 01)`

```
ST4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>], <Xm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant boolean wback = TRUE;
constant boolean nontemporal = FALSE;
```

Assembler Symbols

<**Vt**> Is the name of the first or only SIMD&FP register to be transferred, encoded in the “Rt” field.

<**Vt2**> Is the name of the second SIMD&FP register to be transferred, encoded as “Rt” plus 1 modulo 32.

<**Vt3**> Is the name of the third SIMD&FP register to be transferred, encoded as “Rt” plus 2 modulo 32.

<**Vt4**> Is the name of the fourth SIMD&FP register to be transferred, encoded as “Rt” plus 3 modulo 32.

<**index**> For the “8-bit”, “8-bit, immediate offset”, and “8-bit, register offset” variants: is the element index, encoded in “Q:S:size”.

For the “16-bit”, “16-bit, immediate offset”, and “16-bit, register offset” variants: is the element index, encoded in “Q:S:size<1>”.

For the “32-bit”, “32-bit, immediate offset”, and “32-bit, register offset” variants: is the element index, encoded in “Q:S”.

For the “64-bit”, “64-bit, immediate offset”, and “64-bit, register offset” variants: is the element index, encoded in “Q”.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<**Xm**> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

Shared Decode

```
bits(2) scale = opcode<2:1>;
constant integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
integer index;

case scale of
    when '11'
        // load and replicate
        if L == '0' || S == '1' then EndOfDecode(Decode_UNDEF);
        scale = size;
        replicate = TRUE;
    when '00'
        index = UInt(Q:S:size);       // B[0-15]
    when '01'
        if size<0> == '1' then EndOfDecode(Decode_UNDEF);
        index = UInt(Q:S:size<1>);    // H[0-7]
    when '10'
        if size<1> == '1' then EndOfDecode(Decode_UNDEF);
        if size<0> == '0' then
            index = UInt(Q:S);        // S[0-3]
        else
            if S == '1' then EndOfDecode(Decode_UNDEF);
            index = UInt(Q);          // D[0-1]
            scale = '11';

constant integer datasize = 64 << UInt(Q);
constant integer esize = 8 << UInt(scale);
```

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(64) address;
bits(64) eaddr;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_STORE, nontemporal, privileged);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

offs = Zeros(64);

if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        eaddr = AddressIncrement(address, offs, accdesc);
        element = Mem[eaddr, ebytes, accdesc];
        // replicate to fill 128- or 64-bit register
        V[t, datasize] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t, 128];
        eaddr = AddressIncrement(address, offs, accdesc);
        // extract from one lane of 128-bit register
        Mem[eaddr, ebytes, accdesc] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = ( t + 1 ) MOD 32;
if wback then
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if m != 31 then
    offs = X[m, 64];
address = AddressAdd(address, offs, accdesc);
if n == 31 then
    SP[64] = address;
else
    X[n, 64] = address;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.313 STADD, STADDL

Atomic add on word or doubleword, without return

This instruction atomically loads a 32-bit word or 64-bit doubleword from memory, adds the value held in a register to it, and stores the result back to memory.

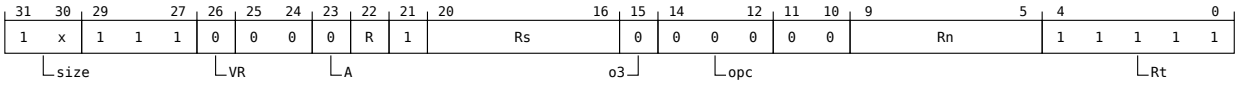
- STADD does not have release semantics.
- STADDL stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDADD, LDADDA, LDADDAL, LDADDL. This means:

- The encodings in this description are named to match the encodings of LDADD, LDADDA, LDADDAL, LDADDL.
- The description of LDADD, LDADDA, LDADDAL, LDADDL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer
(FEAT_LSE)



Encoding for the 32-bit no memory ordering variant Applies when `(size == 10 && R == 0)`

```
STADD <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDADD <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 32-bit release variant Applies when `(size == 10 && R == 1)`

```
STADDL <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDADDL <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 64-bit no memory ordering variant Applies when `(size == 11 && R == 0)`

```
STADD <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDADD <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the 64-bit release variant Applies when `(size == 11 && R == 1)`

```
STADDL <Xs>, [<Xn|SP>]
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

is equivalent to

```
LDADDL <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<**Ws**> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

<**Xn**|**SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<**Xs**> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the “Rs” field.

Operation

The description of LDADD, LDADDA, LDADDAL, LDADDL gives the operational pseudocode for this instruction.

Operational Information

The description of LDADD, LDADDA, LDADDAL, LDADDL gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.314 STADDB, STADDLB

Atomic add on byte, without return

This instruction atomically loads an 8-bit byte from memory, adds the value held in a register to it, and stores the result back to memory.

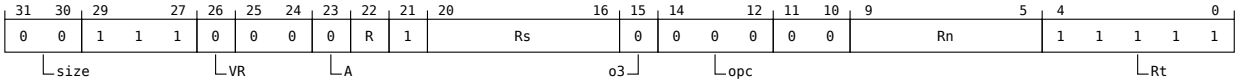
- `STADDB` does not have release semantics.
- `STADDLB` stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDADDB, LDADDAB, LDADDALB, LDADDLB. This means:

- The encodings in this description are named to match the encodings of LDADDB, LDADDAB, LDADDALB, LDADDLB.
- The description of LDADDB, LDADDAB, LDADDALB, LDADDLB gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer
(FEAT_LSE)



Encoding for the No memory ordering variant Applies when `(R == 0)`

```
STADDB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDADDB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the Release variant Applies when `(R == 1)`

```
STADDLB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDADDLB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<**Ws**> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDADDB, LDADDAB, LDADDALB, LDADDLB gives the operational pseudocode for this instruction.

Operational Information

The description of LDADDB, LDADDAB, LDADDALB, LDADDLB gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.315 STADDH, STADDLH

Atomic add on halfword, without return

This instruction atomically loads a 16-bit halfword from memory, adds the value held in a register to it, and stores the result back to memory.

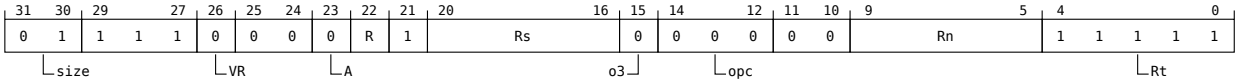
- STADDH does not have release semantics.
- STADDLH stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about addressing modes, see Load/Store addressing modes.

This is an alias of LDADDH, LDADDAH, LDADDALH, LDADDLH. This means:

- The encodings in this description are named to match the encodings of LDADDH, LDADDAH, LDADDALH, LDADDLH.
- The description of LDADDH, LDADDAH, LDADDALH, LDADDLH gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer
(FEAT_LSE)



Encoding for the No memory ordering variant Applies when (R == 0)

```
STADDH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDADDH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Encoding for the Release variant Applies when (R == 1)

```
STADDLH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDADDLH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<**Ws**> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDADDH, LDADDAH, LDADDALH, LDADDLH gives the operational pseudocode for this instruction.

Operational Information

The description of LDADDH, LDADDAH, LDADDALH, LDADDLH gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

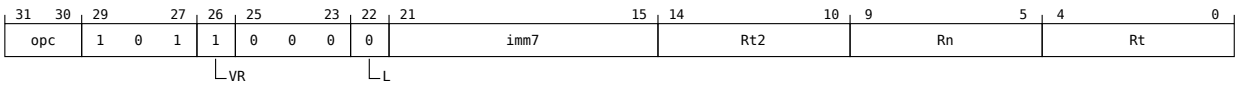
C7.2.316 STNP (SIMD&FP)

Store pair of SIMD&FP registers, with non-temporal hint

This instruction stores a pair of SIMD&FP registers to memory, issuing a hint to the memory system that the access is non-temporal. The address used for the store is calculated from an address from a base register value and an immediate offset. For information about non-temporal pair instructions, see Load/Store SIMD and Floating-point non-temporal pair.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Signed offset
(FEAT_FP)



Encoding for the 32-bit variant Applies when `(opc == 00)`

```
STNP <St1>, <St2>, [<Xn|SP>{, #<imm>}]
```

Encoding for the 64-bit variant Applies when `(opc == 01)`

```
STNP <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}]
```

Encoding for the 128-bit variant Applies when `(opc == 10)`

```
STNP <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}]
```

Decode for all variants of this encoding

```
// Empty.
```

Assembler Symbols

<**St1**> Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the “Rt” field.

<**St2**> Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the “Rt2” field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<**imm**> For the “32-bit” variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the “imm7” field as <imm>/4.

For the “64-bit” variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the “imm7” field as <imm>/8.

For the “128-bit” variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the “imm7” field as <imm>/16.

<**Dt1**> Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the “Rt” field.

<**Dt2**> Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the “Rt2” field.

<**Qt1**> Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the “Rt” field.

<**Qt2**> Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the “Rt2” field.

Shared Decode

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
constant integer t = UInt(Rt);
constant integer t2 = UInt(Rt2);
constant integer n = UInt(Rn);
constant boolean nontemporal = TRUE;
constant integer scale = 2 + (UInt(opc));
constant integer datasize = 8 << scale;
constant bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
bits(64) address;
constant integer dbytes = datasize DIV 8;

constant boolean privileged = PSTATE.EL != EL0;
constant boolean ispair = TRUE;
constant AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_STORE, nontemporal,
                                                       privileged, ispair);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

bits(2*datasize) data;
if BigEndian(accdesc.acctype) then
    data = V[t, datasize] : V[t2, datasize];
else
    data = V[t2, datasize] : V[t, datasize];
Mem[address, 2*dbytes, accdesc] = data;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.317 STP (SIMD&FP)

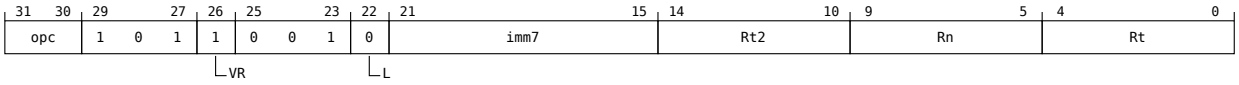
Store pair of SIMD&FP registers

This instruction stores a pair of SIMD&FP registers to memory. The address used for the store is calculated from a base register value and an immediate offset.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 3 classes: Post-index , Pre-index and Signed offset

Post-index
(FEAT_FP)



Encoding for the 32-bit variant Applies when `(opc == 00)`

```
STP <St1>, <St2>, [<Xn|SP>], #<imm>
```

Encoding for the 64-bit variant Applies when `(opc == 01)`

```
STP <Dt1>, <Dt2>, [<Xn|SP>], #<imm>
```

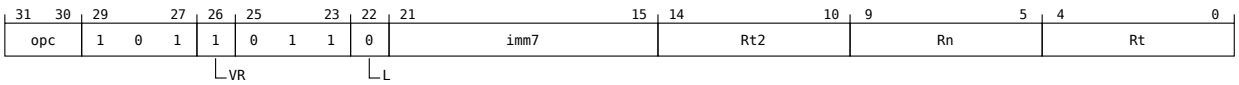
Encoding for the 128-bit variant Applies when `(opc == 10)`

```
STP <Qt1>, <Qt2>, [<Xn|SP>], #<imm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);

constant boolean wback = TRUE;
constant boolean postindex = TRUE;
```

Pre-index
(FEAT_FP)



Encoding for the 32-bit variant Applies when `(opc == 00)`

```
STP <St1>, <St2>, [<Xn|SP>, #<imm>]!
```

Encoding for the 64-bit variant Applies when `(opc == 01)`

```
STP <Dt1>, <Dt2>, [<Xn|SP>, #<imm>]!
```

Encoding for the 128-bit variant Applies when `(opc == 10)`

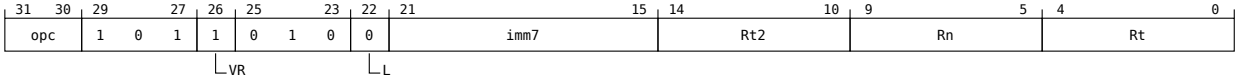
```
STP <Qt1>, <Qt2>, [<Xn|SP>, #<imm>]!
```

Decode for all variants of this encoding

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);

constant boolean wback = TRUE;
constant boolean postindex = FALSE;
```

Signed offset
(FEAT_FP)



Encoding for the 32-bit variant Applies when `(opc == 00)`

```
STP <St1>, <St2>, [<Xn|SP>{, #<imm>}]
```

Encoding for the 64-bit variant Applies when `(opc == 01)`

```
STP <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}]
```

Encoding for the 128-bit variant Applies when `(opc == 10)`

```
STP <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);

constant boolean wback = FALSE;
constant boolean postindex = FALSE;
```

Assembler Symbols

<**St1**> Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the “Rt” field.

<**St2**> Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the “Rt2” field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<**imm**> For the “Post-index 32-bit” and “Pre-index 32-bit” variants: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the “imm7” field as <imm>/4.

For the “Post-index 64-bit” and “Pre-index 64-bit” variants: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the “imm7” field as <imm>/8.

For the “Post-index 128-bit” and “Pre-index 128-bit” variants: is the signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, encoded in the “imm7” field as <imm>/16.

For the “Signed offset 32-bit” variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the “imm7” field as <imm>/4.

For the “Signed offset 64-bit” variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the “imm7” field as <imm>/8.

For the “Signed offset 128-bit” variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the “imm7” field as <imm>/16.

<**Dt1**> Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the “Rt” field.

<**Dt2**> Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the “Rt2” field.

<**Qt1**> Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the “Rt” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<Qt2> Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer t2 = UInt(Rt2);
constant integer n = UInt(Rn);
constant boolean nontemporal = FALSE;
constant integer scale = 2 + (UInt(opc));
constant integer datasize = 8 << scale;
constant bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
bits(64) address;
constant integer dbytes = datasize DIV 8;

constant boolean privileged = PSTATE.EL != EL0;
constant boolean ispair = TRUE;
constant AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_STORE, nontemporal,
                                                       privileged, ispair);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

if !postindex then
    address = AddressAdd(address, offset, accdesc);

bits(2*datasize) data;
if BigEndian(accdesc.acctype) then
    data = V[t, datasize] : V[t2, datasize];
else
    data = V[t2, datasize] : V[t, datasize];

Mem[address, 2*dbytes, accdesc] = data;

if wback then
    if postindex then
        address = AddressAdd(address, offset, accdesc);
    if n == 31 then
        SP[64] = address;
    else
        X[n, 64] = address;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.318 STR (immediate, SIMD&FP)

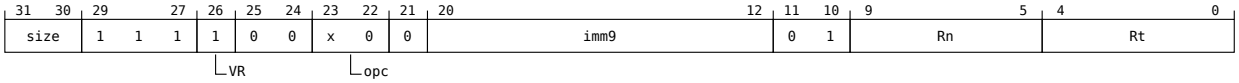
Store SIMD&FP register (immediate offset)

This instruction stores a single SIMD&FP register to memory. The address that is used for the store is calculated from a base register value and an immediate offset.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 3 classes: Post-index , Pre-index and Unsigned offset

Post-index
(FEAT_FP)



Encoding for the 8-bit variant Applies when `(size == 00 && opc == 00)`

```
STR <Bt>, [<Xn|SP>], #<simm>
```

Encoding for the 16-bit variant Applies when `(size == 01 && opc == 00)`

```
STR <Ht>, [<Xn|SP>], #<simm>
```

Encoding for the 32-bit variant Applies when `(size == 10 && opc == 00)`

```
STR <St>, [<Xn|SP>], #<simm>
```

Encoding for the 64-bit variant Applies when `(size == 11 && opc == 00)`

```
STR <Dt>, [<Xn|SP>], #<simm>
```

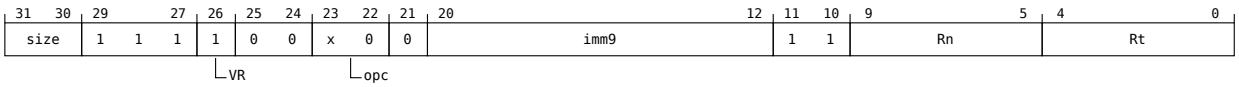
Encoding for the 128-bit variant Applies when `(size == 00 && opc == 10)`

```
STR <Qt>, [<Xn|SP>], #<simm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if opc<1> == '1' && size != '00' then EndOfDecode(Decode_UNDEF);
constant integer scale = if opc<1> == '1' then 4 else UInt(size);
constant boolean wback = TRUE;
constant boolean postindex = TRUE;
constant bits(64) offset = SignExtend(imm9, 64);
```

Pre-index
(FEAT_FP)



Encoding for the 8-bit variant Applies when `(size == 00 && opc == 00)`

```
STR <Bt>, [<Xn|SP>, #<simm>]!
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encoding for the 16-bit variant Applies when `(size == 01 && opc == 00)`

```
STR <Ht>, [<Xn|SP>, #<simm>]!
```

Encoding for the 32-bit variant Applies when `(size == 10 && opc == 00)`

```
STR <St>, [<Xn|SP>, #<simm>]!
```

Encoding for the 64-bit variant Applies when `(size == 11 && opc == 00)`

```
STR <Dt>, [<Xn|SP>, #<simm>]!
```

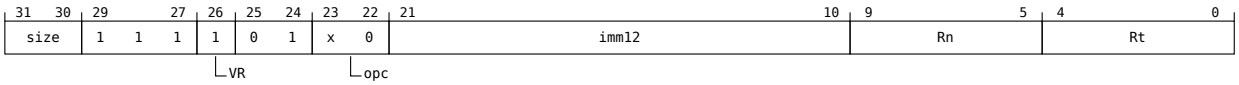
Encoding for the 128-bit variant Applies when `(size == 00 && opc == 10)`

```
STR <Qt>, [<Xn|SP>, #<simm>]!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if opc<1> == '1' && size != '00' then EndOfDecode(Decode_UNDEF);
constant integer scale = if opc<1> == '1' then 4 else UInt(size);
constant boolean wback = TRUE;
constant boolean postindex = FALSE;
constant bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset
(FEAT_FP)



Encoding for the 8-bit variant Applies when `(size == 00 && opc == 00)`

```
STR <Bt>, [<Xn|SP>{, #<pimm>}]
```

Encoding for the 16-bit variant Applies when `(size == 01 && opc == 00)`

```
STR <Ht>, [<Xn|SP>{, #<pimm>}]
```

Encoding for the 32-bit variant Applies when `(size == 10 && opc == 00)`

```
STR <St>, [<Xn|SP>{, #<pimm>}]
```

Encoding for the 64-bit variant Applies when `(size == 11 && opc == 00)`

```
STR <Dt>, [<Xn|SP>{, #<pimm>}]
```

Encoding for the 128-bit variant Applies when `(size == 00 && opc == 10)`

```
STR <Qt>, [<Xn|SP>{, #<pimm>}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if opc<1> == '1' && size != '00' then EndOfDecode(Decode_UNDEF);
constant integer scale = if opc<1> == '1' then 4 else UInt(size);
constant boolean wback = FALSE;
constant boolean postindex = FALSE;
constant bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<**Bt**> Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<**simm**> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.

<**Ht**> Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<**St**> Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<**Dt**> Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<**Qt**> Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<**pimm**> For the "8-bit" variant: is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For the "16-bit" variant: is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2.

For the "32-bit" variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4.

For the "64-bit" variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.

For the "128-bit" variant: is the optional positive immediate byte offset, a multiple of 16 in the range 0 to 65520, defaulting to 0 and encoded in the "imm12" field as <pimm>/16.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer datasize = 8 << scale;
constant boolean nontemporal = FALSE;
```

Operation

```
AArch64.CheckFPEnabled();
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_STORE, nontemporal, privileged);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

if !postindex then
    address = AddressAdd(address, offset, accdesc);

Mem[address, datasize DIV 8, accdesc] = V[t, datasize];

if wback then
    if postindex then
        address = AddressAdd(address, offset, accdesc);
    if n == 31 then
        SP[64] = address;
    else
        X[n, 64] = address;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

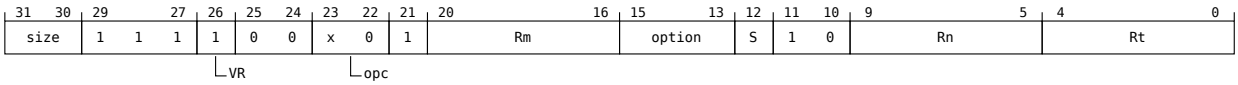
C7.2.319 STR (register, SIMD&FP)

Store SIMD&FP register (register offset)

This instruction stores a single SIMD&FP register to memory. The address that is used for the store is calculated from a base register value and an offset register value. The offset can be optionally shifted and extended.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

SIMD&FP registers
(FEAT_FP)



Encoding for the 8-bit with extended register offset variant Applies when `(size == 00 && opc == 00 && option != 011)`

```
STR <Bt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

Encoding for the 8-bit with shifted register offset variant Applies when `(size == 00 && opc == 00 && option == 011)`

```
STR <Bt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

Encoding for the 16-bit variant Applies when `(size == 01 && opc == 00)`

```
STR <Ht>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Encoding for the 32-bit variant Applies when `(size == 10 && opc == 00)`

```
STR <St>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Encoding for the 64-bit variant Applies when `(size == 11 && opc == 00)`

```
STR <Dt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Encoding for the 128-bit variant Applies when `(size == 00 && opc == 10)`

```
STR <Qt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if option<1> == '0' then EndOfDecode(Decode_UNDEF);             // sub-word index
if opc<1> == '1' && size != '00' then EndOfDecode(Decode_UNDEF);
constant integer scale = if opc<1> == '1' then 4 else UInt(size);
constant ExtendType extend_type = DecodeRegExtend(option);
constant integer shift = if S == '1' then scale else 0;
```

Assembler Symbols

<**Bt**> Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

<**Xn**|**SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<**Wm**> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the “Rm” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**Xm**> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<**extend**> For the "8-bit with extended register offset" variant: is the index extend specifier, encoded in "option":

option	<extend>
010	UXTW
110	SXTW
111	SXTX

For the "128-bit", "16-bit", "32-bit", and "64-bit" variants: is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted, encoded in "option":

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<**amount**> For the "8-bit with extended register offset" and "8-bit with shifted register offset" variants: is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.

For the "16-bit" variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

S	<amount>
0	#0
1	#1

For the "32-bit" variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

S	<amount>
0	#0
1	#2

For the "64-bit" variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

S	<amount>
0	#0
1	#3

For the "128-bit" variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

S	<amount>
0	#0
1	#4

<**Ht**> Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<**St**> Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<**Dt**> Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<**Qt**> Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer datasize = 8 << scale;
constant boolean nontemporal = FALSE;
```

Operation

```
AArch64.CheckFPEnabled();
constant bits(64) offset = ExtendReg(m, extend_type, shift, 64);
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_STORE, nontemporal, privileged);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

Mem[address, datasize DIV 8, accdesc] = V[t, datasize];
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

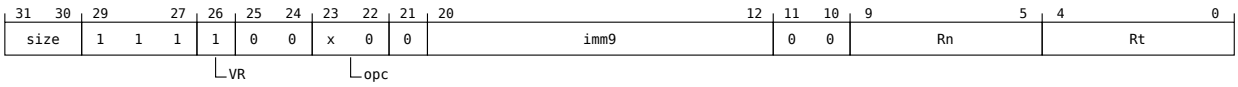
C7.2.320 STUR (SIMD&FP)

Store SIMD&FP register (unscaled offset)

This instruction stores a single SIMD&FP register to memory. The address that is used for the store is calculated from a base register value and an optional immediate offset.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Unscaled offset
(FEAT_FP)



Encoding for the 8-bit variant Applies when `(size == 00 && opc == 00)`

```
STUR <Bt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 16-bit variant Applies when `(size == 01 && opc == 00)`

```
STUR <Ht>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 32-bit variant Applies when `(size == 10 && opc == 00)`

```
STUR <St>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 64-bit variant Applies when `(size == 11 && opc == 00)`

```
STUR <Dt>, [<Xn|SP>{, #<simm>}]
```

Encoding for the 128-bit variant Applies when `(size == 00 && opc == 10)`

```
STUR <Qt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if opc<1> == '1' && size != '00' then EndOfDecode(Decode_UNDEF);
constant integer scale = if opc<1> == '1' then 4 else UInt(size);
constant bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<**Bt**> Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

<**Xn|SP**> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<**simm**> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the “imm9” field.

<**Ht**> Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

<**St**> Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

<**Dt**> Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

<**Qt**> Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Shared Decode

```
constant integer t = UInt(Rt);
constant integer n = UInt(Rn);
constant integer datasize = 8 << scale;
constant boolean nontemporal = FALSE;
```

Operation

```
AArch64.CheckFPEnabled();
bits(64) address;

constant boolean privileged = PSTATE.EL != EL0;
constant AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_STORE, nontemporal, privileged);

if n == 31 then
    CheckSPAlignment();
    address = SP[64];
else
    address = X[n, 64];

address = AddressAdd(address, offset, accdesc);

Mem[address, datasize DIV 8, accdesc] = V[t, datasize];
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.321 SUB (vector)

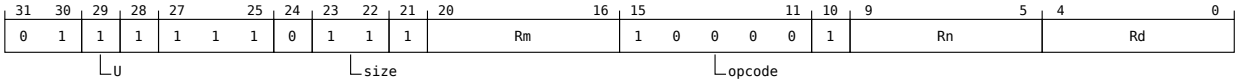
Subtract (vector)

This instruction subtracts each vector element in the second source SIMD&FP register from the corresponding vector element in the first source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



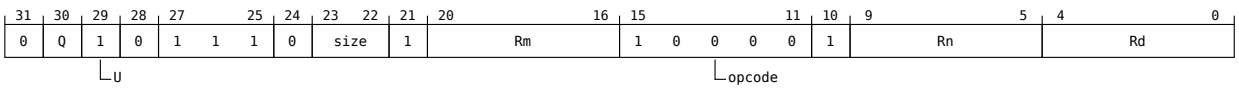
Encoding

```
SUB D<d>, D<n>, D<m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size != '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
SUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**d**> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<**n**> Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<**m**> Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = element1 - element2;
V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.322 SUBHN, SUBHN2

Subtract returning high narrow

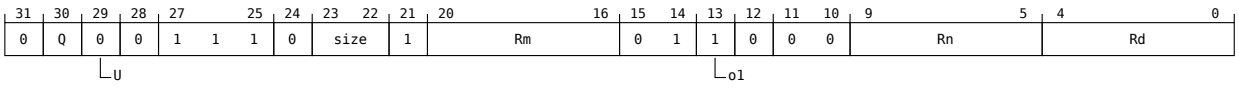
This instruction subtracts each vector element in the second source SIMD&FP register from the corresponding vector element in the first source SIMD&FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are signed integer values.

The results are truncated. For rounded results, see RSUBHN.

The SUBHN instruction writes the vector to the lower half of the destination register and clears the upper half. The SUBHN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers, not all the same type
(FEAT_AdvSIMD)



Encoding

```
SUBHN{2} <Vd>.<Tb>, <Vn>.<Ta>, <Vm>.<Ta>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;
constant boolean round = FALSE;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Tb**> Is an arrangement specifier, encoded in “size:Q”:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Ta**> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(2*datasize) operand1 = V[n, 2*datasize];
constant bits(2*datasize) operand2 = V[m, 2*datasize];
bits(datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, 2*esize]);
    element2 = UInt(Elem[operand2, e, 2*esize]);
    sum = element1 - element2;
    sum = RShr(sum, esize, round);
    Elem[result, e, esize] = sum<esize-1:0>;

Vpart[d, part, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.323 SUQADD

Signed saturating accumulate of unsigned value

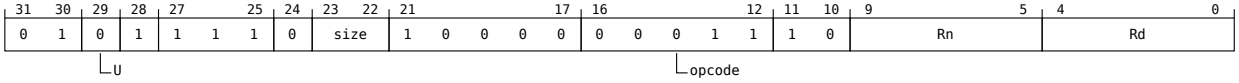
This instruction adds the unsigned integer values of the vector elements in the source SIMD&FP register to corresponding signed integer values of the vector elements in the destination SIMD&FP register, and writes the resulting signed integer values to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



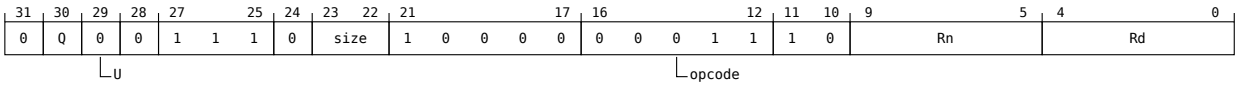
Encoding

```
SUQADD <V><d>, <V><n>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
constant boolean unsigned = FALSE;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
SUQADD <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
constant boolean unsigned = FALSE;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<**V**> Is a width specifier, encoded in "size":

size	<V>
00	B
01	H
10	S
11	D

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**n**> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(datasize) result;

constant bits(datasize) operand2 = V[d, datasize];
integer op1;
integer op2;
boolean sat;

for e = 0 to elements-1
    op1 = UInt(Elem[operand, e, esize]);
    op2 = SInt(Elem[operand2, e, esize]);
    (Elem[result, e, esize], sat) = SatQ(op1 + op2, esize, unsigned);
    if sat then FPSR.QC = '1';
V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.324 SXTL, SXTL2

Signed extend long

This instruction duplicates each vector element in the lower or upper half of the source SIMD&FP register into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are signed integer values.

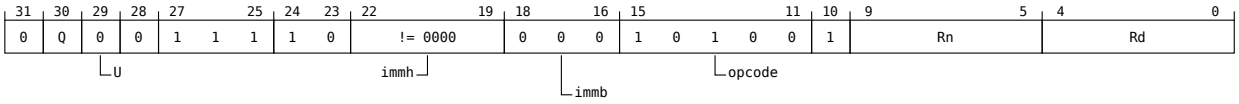
The `SXTL` instruction extracts the source vector from the lower half of the source register. The `SXTL2` instruction extracts the source vector from the upper half of the source register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of SSHLL, SSHLL2. This means:

- The encodings in this description are named to match the encodings of SSHLL, SSHLL2.
- The description of SSHLL, SSHLL2 gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Vector
(FEAT_AdvSIMD)



Encoding

```
SXTL{2} <Vd>.<Ta>, <Vn>.<Tb>
```

is equivalent to

```
SSHLL{2} <Vd>.<Ta>, <Vn>.<Tb>, #0
```

and is the preferred disassembly when `BitCount(immh) == 1`.

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

Q	2
0	[absent]
1	[present]

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**Ta**> Is an arrangement specifier, encoded in "immh":

immh	<Ta>
0001	8H
001x	4S

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

immh	<Ta>
01xx	2D
1xxx	RESERVED

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<**Tb**> Is an arrangement specifier, encoded in "immh:Q":

immh	Q	<Tb>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

Operation

The description of SSHLL, SSHLL2 gives the operational pseudocode for this instruction.

Operational Information

The description of SSHLL, SSHLL2 gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

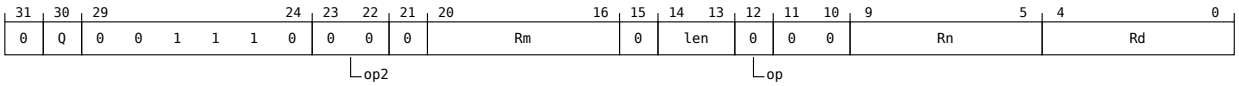
C7.2.325 TBL

Table vector lookup

This instruction reads each value from the vector elements in the index source SIMD&FP register, uses each result as an index to perform a lookup in a table of bytes that is described by one to four source table SIMD&FP registers, places the lookup result in a vector, and writes the vector to the destination SIMD&FP register. If an index is out of range for the table, the result for that lookup is 0. If more than one source register is used to describe the table, the first source register describes the lowest bytes of the table.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Advanced SIMD
(FEAT_AdvSIMD)



Encoding for the Single register table variant Applies when `(len == 00)`

```
TBL <Vd>.<Ta>, { <Vn>.16B }, <Vm>.<Ta>
```

Encoding for the Two register table variant Applies when `(len == 01)`

```
TBL <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B }, <Vm>.<Ta>
```

Encoding for the Three register table variant Applies when `(len == 10)`

```
TBL <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B }, <Vm>.<Ta>
```

Encoding for the Four register table variant Applies when `(len == 11)`

```
TBL <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B, <Vn+3>.16B }, <Vm>.<Ta>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV 8;
constant integer regs = UInt(len) + 1;
constant boolean is_tbl = (op == '0');
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**Ta**> Is an arrangement specifier, encoded in "Q":

Q	<Ta>
0	8B
1	16B

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**Vn**> For the “Single register table” variant: is the name of the SIMD&FP table register, encoded in the “Rn” field.

For the “Four register table”, “Three register table”, and “Two register table” variants: is the name of the first SIMD&FP table register, encoded in the “Rn” field.

<**Vm**> Is the name of the SIMD&FP index register, encoded in the “Rm” field.

<**Vn+1**> Is the name of the second SIMD&FP table register, encoded as “Rn” plus 1 modulo 32.

<**Vn+2**> Is the name of the third SIMD&FP table register, encoded as “Rn” plus 2 modulo 32.

<**Vn+3**> Is the name of the fourth SIMD&FP table register, encoded as “Rn” plus 3 modulo 32.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) indices = V[m, datasize];
bits(128*regs) table = Zeros(128*regs);
bits(datasize) result;
integer index;

// Create table from registers
for i = 0 to regs - 1
    Elem[table, i, 128] = V[(n+i) MOD 32, 128];

result = if is_tbl then Zeros(datasize) else V[d, datasize];
for i = 0 to elements - 1
    index = UInt(Elem[indices, i, 8]);
    if index < 16 * regs then
        Elem[result, i, 8] = Elem[table, index, 8];

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

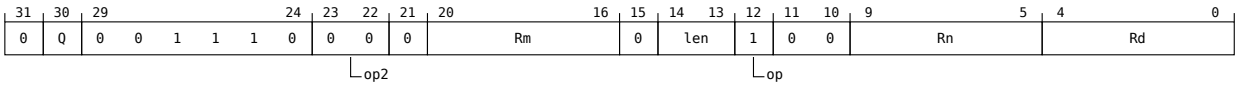
C7.2.326 TBX

Table vector lookup extension

This instruction reads each value from the vector elements in the index source SIMD&FP register, uses each result as an index to perform a lookup in a table of bytes that is described by one to four source table SIMD&FP registers, places the lookup result in a vector, and writes the vector to the destination SIMD&FP register. If an index is out of range for the table, the existing value in the vector element of the destination register is left unchanged. If more than one source register is used to describe the table, the first source register describes the lowest bytes of the table.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Advanced SIMD
(FEAT_AdvSIMD)



Encoding for the Single register table variant Applies when `(len == 00)`

```
TBX <Vd>.<Ta>, { <Vn>.16B }, <Vm>.<Ta>
```

Encoding for the Two register table variant Applies when `(len == 01)`

```
TBX <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B }, <Vm>.<Ta>
```

Encoding for the Three register table variant Applies when `(len == 10)`

```
TBX <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B }, <Vm>.<Ta>
```

Encoding for the Four register table variant Applies when `(len == 11)`

```
TBX <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B, <Vn+3>.16B }, <Vm>.<Ta>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV 8;
constant integer regs = UInt(len) + 1;
constant boolean is_tbl = (op == '0');
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**Ta**> Is an arrangement specifier, encoded in "Q":

Q	<Ta>
0	8B
1	16B

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**Vn**> For the “Single register table” variant: is the name of the SIMD&FP table register, encoded in the “Rn” field.

For the “Four register table”, “Three register table”, and “Two register table” variants: is the name of the first SIMD&FP table register, encoded in the “Rn” field.

<**Vm**> Is the name of the SIMD&FP index register, encoded in the “Rm” field.

<**Vn+1**> Is the name of the second SIMD&FP table register, encoded as “Rn” plus 1 modulo 32.

<**Vn+2**> Is the name of the third SIMD&FP table register, encoded as “Rn” plus 2 modulo 32.

<**Vn+3**> Is the name of the fourth SIMD&FP table register, encoded as “Rn” plus 3 modulo 32.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) indices = V[m, datasize];
bits(128*regs) table = Zeros(128*regs);
bits(datasize) result;
integer index;

// Create table from registers
for i = 0 to regs - 1
    Elem[table, i, 128] = V[(n+i) MOD 32, 128];

result = if is_tbl then Zeros(datasize) else V[d, datasize];
for i = 0 to elements - 1
    index = UInt(Elem[indices, i, 8]);
    if index < 16 * regs then
        Elem[result, i, 8] = Elem[table, index, 8];

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.327 TRN1

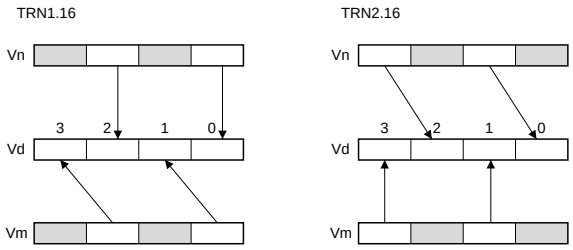
Transpose vectors (primary)

This instruction reads corresponding even-numbered vector elements from the two source SIMD&FP registers, starting at zero, places each result into consecutive elements of a vector, and writes the vector to the destination SIMD&FP register. Vector elements from the first source register are placed into even-numbered elements of the destination vector, starting at zero, while vector elements from the second source register are placed into odd-numbered elements of the destination vector.

Note

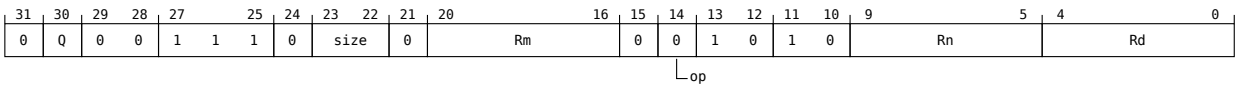
By using this instruction with `TRN2`, a 2 x 2 matrix can be transposed.

The following figure shows an example of the operation of TRN1 and TRN2 halfword operations where Q = 0.



Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Advanced SIMD
(FEAT_AdvSIMD)



Encoding

```
TRN1 <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
constant integer part = UInt(op);
constant integer pairs = elements DIV 2;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**T**> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

for p = 0 to pairs-1
    Elem[result, 2*p+0, esize] = Elem[operand1, 2*p+part, esize];
    Elem[result, 2*p+1, esize] = Elem[operand2, 2*p+part, esize];

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.328 TRN2

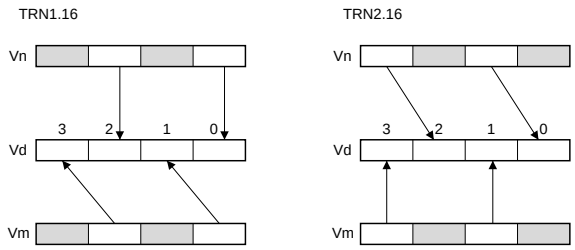
Transpose vectors (secondary)

This instruction reads corresponding odd-numbered vector elements from the two source SIMD&FP registers, places each result into consecutive elements of a vector, and writes the vector to the destination SIMD&FP register. Vector elements from the first source register are placed into even-numbered elements of the destination vector, starting at zero, while vector elements from the second source register are placed into odd-numbered elements of the destination vector.

Note

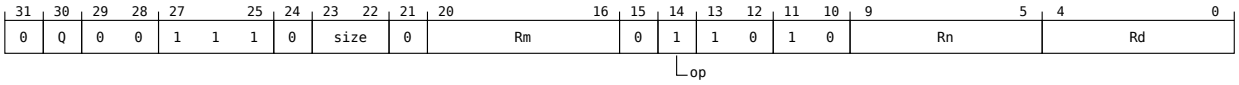
By using this instruction with `TRN1`, a 2 x 2 matrix can be transposed.

The following figure shows an example of the operation of TRN1 and TRN2 halfword operations where Q = 0.



Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Advanced SIMD
(FEAT_AdvSIMD)



Encoding

`TRN2 <Vd>.<T>, <Vn>.<T>, <Vm>.<T>`

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
constant integer part = UInt(op);
constant integer pairs = elements DIV 2;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T> Is an arrangement specifier, encoded in “size:Q”:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

for p = 0 to pairs-1
    Elem[result, 2*p+0, esize] = Elem[operand1, 2*p+part, esize];
    Elem[result, 2*p+1, esize] = Elem[operand2, 2*p+part, esize];

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

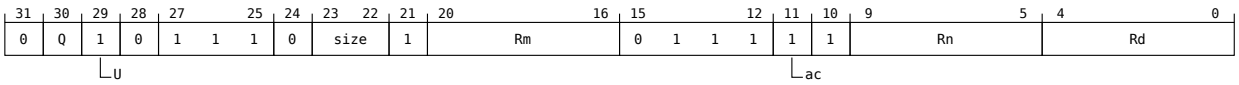
C7.2.329 UABA

Unsigned absolute difference and accumulate

This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, and accumulates the absolute values of the results into the elements of the vector of the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type
(FEAT_AdvSIMD)



Encoding

```
UABA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result = V[d, datasize];
integer element1;
integer element2;
bits(esize) absdiff;

for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    element2 = UInt(Elem[operand2, e, esize]);
    absdiff = Abs(element1 - element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.330 UABAL, UABAL2

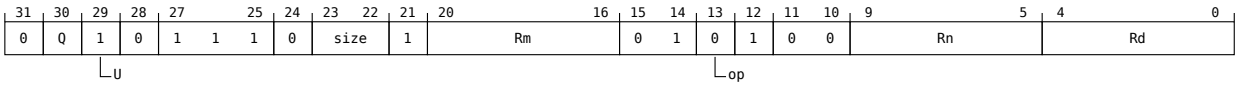
Unsigned absolute difference and accumulate long

This instruction subtracts the vector elements in the lower or upper half of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, and accumulates the absolute values of the results into the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

The `UABAL` instruction extracts each source vector from the lower half of each source register. The `UABAL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers, not all the same type
(FEAT_AdvSIMD)



Encoding

`UABAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>`

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "size":

size	<Ta>
00	8H

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<Ta>
01	4S
10	2D
11	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Tb**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = Vpart[n, part, datasize];
constant bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result = V[d, 2*datasize];
integer element1;
integer element2;
bits(2*esize) absdiff;

for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    element2 = UInt(Elem[operand2, e, esize]);
    absdiff = Abs(element1-element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + absdiff;
V[d, 2*datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

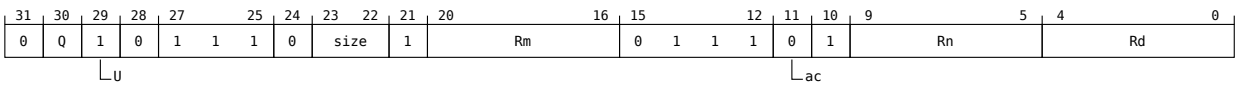
C7.2.331 UABD

Unsigned absolute difference (vector)

This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, places the absolute values of the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type
(FEAT_AdvSIMD)



Encoding

```
UABD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result = Zeros(datasize);
integer element1;
integer element2;
bits(esize) absdiff;

for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    element2 = UInt(Elem[operand2, e, esize]);
    absdiff = Abs(element1 - element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.332 UABDL, UABDL2

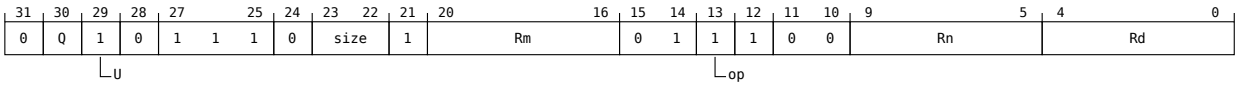
Unsigned absolute difference long

This instruction subtracts the vector elements in the lower or upper half of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, places the absolute value of the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

The `UABDL` instruction extracts each source vector from the lower half of each source register. The `UABDL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers, not all the same type
(FEAT_AdvSIMD)



Encoding

```
UABDL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "size":

size	<Ta>
00	8H

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<Ta>
01	4S
10	2D
11	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Tb**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = Vpart[n, part, datasize];
constant bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result = Zeros(2*datasize);
integer element1;
integer element2;
bits(2*esize) absdiff;

for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    element2 = UInt(Elem[operand2, e, esize]);
    absdiff = Abs(element1-element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + absdiff;
V[d, 2*datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

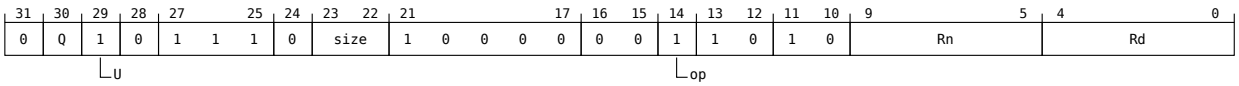
C7.2.333 UADALP

Unsigned add and accumulate long pairwise

This instruction adds pairs of adjacent unsigned integer values from the vector in the source SIMD&FP register and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector
(FEAT_AdvSIMD)



Encoding

```
UADALP <Vd>.<Ta>, <Vn>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV (2 * esize);
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Ta**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Ta>
00	0	4H
00	1	8H
01	0	2S
01	1	4S
10	0	1D
10	1	2D
11	x	RESERVED

<**Vn**> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<**Tb**> Is an arrangement specifier, encoded in “size:Q”:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(datasize) result  = V[d, datasize];

bits(2*esize) sum;
integer op1;
integer op2;

for e = 0 to elements-1
    op1 = UInt(Elem[operand, 2*e+0, esize]);
    op2 = UInt(Elem[operand, 2*e+1, esize]);
    sum = (op1+op2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.334 UADDL, UADDL2

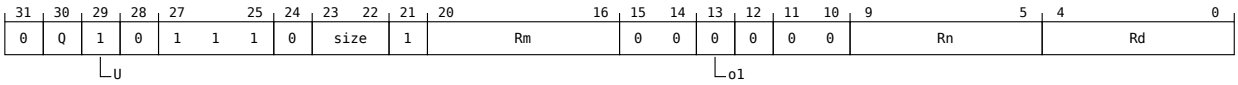
Unsigned add long (vector)

This instruction adds each vector element in the lower or upper half of the first source SIMD&FP register to the corresponding vector element of the second source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

The `UADDL` instruction extracts each source vector from the lower half of each source register. The `UADDL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers, not all the same type
(FEAT_AdvSIMD)



Encoding

```
UADDL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "size":

size	<Ta>
00	8H

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<Ta>
01	4S
10	2D
11	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Tb**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = Vpart[n, part, datasize];
constant bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    element2 = UInt(Elem[operand2, e, esize]);
    sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d, 2*datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

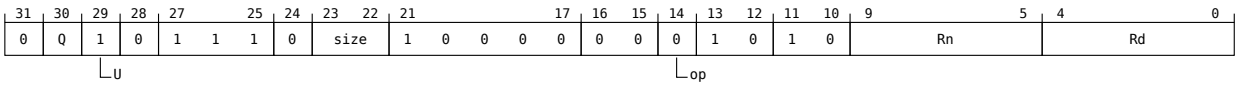
C7.2.335 UADDLP

Unsigned add long pairwise

This instruction adds pairs of adjacent unsigned integer values from the vector in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector
(FEAT_AdvSIMD)



Encoding

```
UADDLP <Vd>.<Ta>, <Vn>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV (2 * esize);
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Ta**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Ta>
00	0	4H
00	1	8H
01	0	2S
01	1	4S
10	0	1D
10	1	2D
11	x	RESERVED

<**Vn**> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<**Tb**> Is an arrangement specifier, encoded in “size:Q”:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(datasize) result;

bits(2*esize) sum;
integer op1;
integer op2;

for e = 0 to elements-1
    op1 = UInt(Elem[operand, 2*e+0, esize]);
    op2 = UInt(Elem[operand, 2*e+1, esize]);
    sum = (op1+op2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = sum;

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

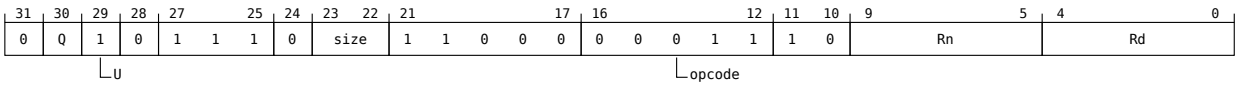
C7.2.336 UADDLV

Unsigned sum long across vector

This instruction adds every vector element in the source SIMD&FP register together, and writes the scalar result to the destination SIMD&FP register. The destination scalar is twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Advanced SIMD
(FEAT_AdvSIMD)



Encoding

```
UADDLV <V><d>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '100' then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;

constant boolean unsigned = TRUE;
```

Assembler Symbols

<**V**> Is the destination width specifier, encoded in “size”:

size	<**V**>
00	H
01	S
10	D
11	RESERVED

<**d**> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<**Vn**> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<**T**> Is an arrangement specifier, encoded in “size:Q”:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	RESERVED
10	1	4S
11	x	RESERVED

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(esize) opelt = Elem[operand, 0, esize];
integer sum = if unsigned then UInt(opelt) else SInt(opelt);

for e = 1 to elements-1
    opelt = Elem[operand, e, esize];
    constant integer element = if unsigned then UInt(opelt) else SInt(opelt);
    sum = sum + element;

V[d, 2*esize] = sum<2*esize-1:0>;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.337 UADDW, UADDW2

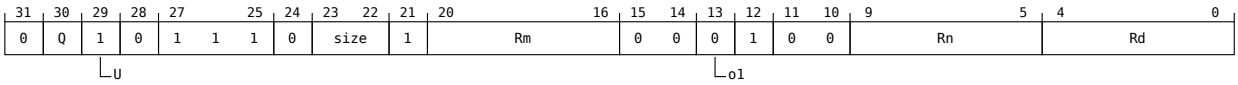
Unsigned add wide

This instruction adds the vector elements of the first source SIMD&FP register to the corresponding vector elements in the lower or upper half of the second source SIMD&FP register, places the result in a vector, and writes the vector to the SIMD&FP destination register. The vector elements of the destination register and the first source register are twice as long as the vector elements of the second source register. All the values in this instruction are unsigned integer values.

The `UADDW` instruction extracts vector elements from the lower half of the second source register. The `UADDW2` instruction extracts vector elements from the upper half of the second source register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers, not all the same type
(FEAT_AdvSIMD)



Encoding

```
UADDW{2} <Vd>.<Ta>, <Vn>.<Ta>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "size":

size	<Ta>
00	8H

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<Ta>
01	4S
10	2D
11	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

<**Tb**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(2*datasize) operand1 = V[n, 2*datasize];
constant bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, 2*esize]);
    element2 = UInt(Elem[operand2, e, esize]);
    sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d, 2*datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.338 UCVTF (vector, fixed-point)

Unsigned fixed-point convert to floating-point (vector)

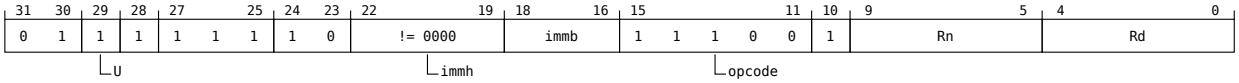
This instruction converts each element in a vector from fixed-point to floating-point using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



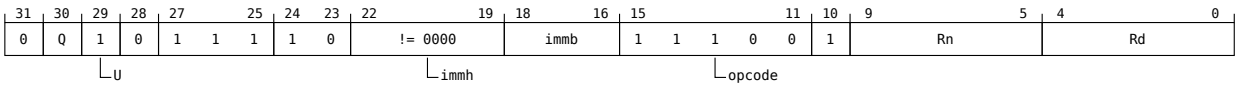
Encoding

```
UCVTF <V><d>, <V><n>, #<fbits>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh IN {'000x'} || (immh IN {'001x'} && !IsFeatureImplemented(FEAT_FP16)) then
    EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = if immh IN {'1xxx'} then 64 else if immh IN {'01xx'} then 32 else 16;
constant integer datasize = esize;
constant integer elements = 1;

constant integer fracbits = (esize * 2) - UInt(immh:immb);
constant boolean unsigned = TRUE;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
UCVTF <Vd>.<T>, <Vn>.<T>, #<fbits>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh IN {'000x'} || (immh IN {'001x'} && !IsFeatureImplemented(FEAT_FP16)) then
    EndOfDecode(Decode_UNDEF);
if immh<3>:Q == '10' then EndOfDecode(Decode_UNDEF);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = if immh IN {'1xxx'} then 64 else if immh IN {'01xx'} then 32 else 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;

constant integer fracbits = (esize * 2) - UInt(immh:immb);
constant boolean unsigned = TRUE;
```

Assembler Symbols

<**V**> Is a width specifier, encoded in "immh":

immh	<V>	Architectural Feature
0001	RESERVED	-
001x	H	FEAT_FP16
01xx	S	-
1xxx	D	-

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**n**> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<**fbits**> For the "Scalar" variant: is the number of fractional bits, in the range 1 to the operand width, encoded in "immh:immb":

immh	<fbits>
0001	RESERVED
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	128 - UInt(immh:immb)

For the "Vector" variant: is the number of fractional bits, in the range 1 to the element width, encoded in "immh:immb":

immh	<fbits>
0001	RESERVED
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	128 - UInt(immh:immb)

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> Is an arrangement specifier, encoded in "immh:Q":

immh	Q	<T>	Architectural Feature
0001	x	RESERVED	-
001x	0	4H	FEAT_FP16

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

immh	Q	<T>	Architectural Feature
001x	1	8H	FEAT_FP16
01xx	0	2S	-
01xx	1	4S	-
1xxx	0	RESERVED	-
1xxx	1	2D	-

<**Vn**> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand  = V[n, datasize];
constant FPRounding rounding = FPRoundingMode(FPCR);

bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FixedToFP(element, fracbits, unsigned, FPCR, rounding, esize);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.339 UCVTF (vector, integer)

Unsigned integer convert to floating-point (vector)

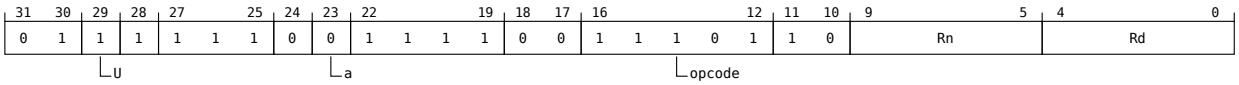
This instruction converts each element in a vector from an unsigned integer value to a floating-point value using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half-precision , Scalar single-precision and double-precision , Vector half-precision and Vector single-precision and double-precision

Scalar half-precision
(FEAT_AdvSIMD && FEAT_FP16)



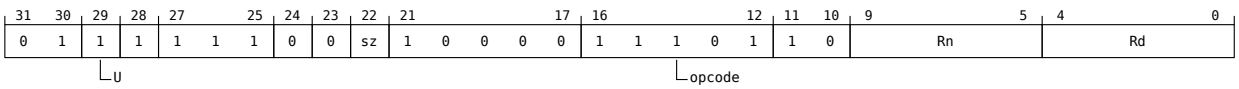
Encoding

```
UCVTF <Hd>, <Hn>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 16;
constant integer datasize = esize;
constant integer elements = 1;

constant boolean unsigned = TRUE;
```

Scalar single-precision and double-precision
(FEAT_AdvSIMD)



Encoding

```
UCVTF <V><d>, <V><n>
```

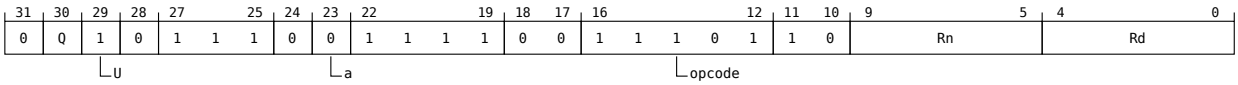
Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 32 << UInt(sz);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer datasize = esize;
constant integer elements = 1;
constant boolean unsigned = TRUE;
```

Vector half-precision (FEAT_AdvSIMD && FEAT_FP16)



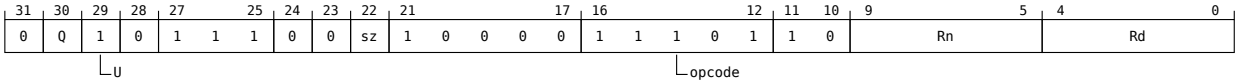
Encoding

```
UCVTF <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) || !IsFeatureImplemented(FEAT_FP16) then
    EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 16;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;

constant boolean unsigned = TRUE;
```

Vector single-precision and double-precision (FEAT_AdvSIMD)



Encoding

```
UCVTF <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

if sz:Q == '10' then EndOfDecode(Decode_UNDEF);
constant integer esize = 32 << UInt(sz);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;

constant boolean unsigned = TRUE;
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Hn**> Is the 16-bit name of the SIMD&FP source register, encoded in the “Rn” field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**V**> Is a width specifier, encoded in "sz":

sz	<V>
0	S
1	D

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**n**> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> For the "Vector half-precision" variant: is an arrangement specifier, encoded in "Q":

Q	<T>
0	4H
1	8H

For the "Vector single-precision and double-precision" variant: is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(datasize) operand = V[n, datasize];

bits(datasize) result;
constant FPRounding rounding = FPRoundingMode(FPCR);
constant integer fracbits = 0;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FixedToFP(element, fracbits, unsigned, FPCR, rounding, esize);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

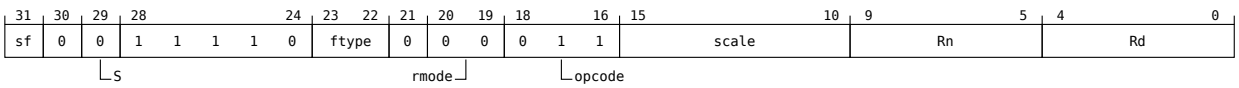
C7.2.340 UCVTF (scalar, fixed-point)

Unsigned fixed-point convert to floating-point (scalar)

This instruction converts the unsigned value in the 32-bit or 64-bit general-purpose source register to a floating-point value using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Encoding for the 32-bit to half-precision variant
(FEAT_FP16) Applies when `(sf == 0 && ftype == 11)`

```
UCVTF <Hd>, <Wn>, #<fbits>
```

Encoding for the 64-bit to half-precision variant
(FEAT_FP16) Applies when `(sf == 1 && ftype == 11)`

```
UCVTF <Hd>, <Xn>, #<fbits>
```

Encoding for the 32-bit to single-precision variant
(FEAT_FP) Applies when `(sf == 0 && ftype == 00)`

```
UCVTF <Sd>, <Wn>, #<fbits>
```

Encoding for the 64-bit to single-precision variant
(FEAT_FP) Applies when `(sf == 1 && ftype == 00)`

```
UCVTF <Sd>, <Xn>, #<fbits>
```

Encoding for the 32-bit to double-precision variant
(FEAT_FP) Applies when `(sf == 0 && ftype == 01)`

```
UCVTF <Dd>, <Wn>, #<fbits>
```

Encoding for the 64-bit to double-precision variant
(FEAT_FP) Applies when `(sf == 1 && ftype == 01)`

```
UCVTF <Dd>, <Xn>, #<fbits>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if ftype == '10' then EndOfDecode(Decode_UNDEF);
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);
if sf == '0' && scale<5> == '0' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer intsize = 32 << UInt(sf);
constant integer decode_fltsize = 8 << UInt(ftype EOR '10');
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer fracbits = 64 - UInt(scale);

constant boolean unsigned = TRUE;
```

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<**Wn**> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

<**fbits**> For the "32-bit to double-precision", "32-bit to half-precision", and "32-bit to single-precision" variants: is the number of bits after the binary point in the fixed-point source, in the range 1 to 32, encoded as 64 minus "scale".

For the "64-bit to double-precision", "64-bit to half-precision", and "64-bit to single-precision" variants: is the number of bits after the binary point in the fixed-point source, in the range 1 to 64, encoded as 64 minus "scale".

<**Xn**> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

<**Sd**> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<**Dd**> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant integer fltsize = decode_fltsize;
constant FPRounding rounding = FPRoundingMode(FPCR);

constant bits(intsize) intval = X[n, intsize];
constant bits(fltsize) fltval = FixedToFP(intval, fracbits, unsigned, FPCR, rounding, fltsize);

V[d, fltsize] = fltval;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

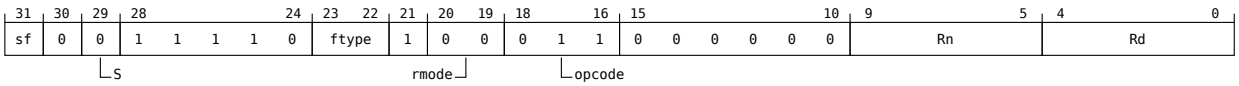
C7.2.341 UCVTF (scalar, integer)

Unsigned integer convert to floating-point (scalar)

This instruction converts the unsigned integer value in the general-purpose source register to a floating-point value using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exceptions and exception traps.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Encoding for the 32-bit to half-precision variant
(FEAT_FP16) Applies when `(sf == 0 && ftype == 11)`

```
UCVTF <Hd>, <Wn>
```

Encoding for the 32-bit to single-precision variant
(FEAT_FP) Applies when `(sf == 0 && ftype == 00)`

```
UCVTF <Sd>, <Wn>
```

Encoding for the 32-bit to double-precision variant
(FEAT_FP) Applies when `(sf == 0 && ftype == 01)`

```
UCVTF <Dd>, <Wn>
```

Encoding for the 64-bit to half-precision variant
(FEAT_FP16) Applies when `(sf == 1 && ftype == 11)`

```
UCVTF <Hd>, <Xn>
```

Encoding for the 64-bit to single-precision variant
(FEAT_FP) Applies when `(sf == 1 && ftype == 00)`

```
UCVTF <Sd>, <Xn>
```

Encoding for the 64-bit to double-precision variant
(FEAT_FP) Applies when `(sf == 1 && ftype == 01)`

```
UCVTF <Dd>, <Xn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FP) then EndOfDecode(Decode_UNDEF);
if ftype == '10' then EndOfDecode(Decode_UNDEF);
if ftype == '11' && !IsFeatureImplemented(FEAT_FP16) then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer intsize = 32 << UInt(sf);
constant integer decode_fltsize = 8 << UInt(ftype EOR '10');
constant boolean unsigned    = TRUE;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<**Hd**> Is the 16-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Wn**> Is the 32-bit name of the general-purpose source register, encoded in the “Rn” field.

<**Sd**> Is the 32-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Dd**> Is the 64-bit name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Xn**> Is the 64-bit name of the general-purpose source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant integer fltsize = decode_fltsize;

constant bits(intsize) intval = X[n, intsize];
constant FPRounding rounding  = FPRoundingMode(FPCR);
constant integer fracbits = 0;
constant bits(fltsize) fltval = FixedToFP(intval, fracbits, unsigned, FPCR, rounding, fltsize);

V[d, fltsize] = fltval;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

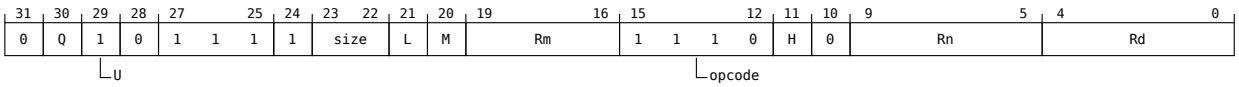
C7.2.342 UDOT (by element)

Dot product unsigned arithmetic (vector, by element)

This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of an indexed 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector
(FEAT_DotProd)



Encoding

```
UDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.4B[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_DotProd) then EndOfDecode(Decode_UNDEF);
if size  != '10' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(M:Rm);
constant integer index = UInt(H:L);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP third source and destination register, encoded in the “Rd” field.

<**Ta**> Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	2S
1	4S

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Tb**> Is an arrangement specifier, encoded in “Q”:

Q	<Tb>
0	8B
1	16B

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “M:Rm” fields.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<**index**> Is the element index, encoded in the “H:L” fields.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(128) operand2 = V[m, 128];
bits(datasize) result = V[d, datasize];
for e = 0 to elements-1
    integer res = 0;
    integer element1, element2;
    for i = 0 to 3
        element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
        element2 = UInt(Elem[operand2, 4 * index + i, esize DIV 4]);

        res = res + element1 * element2;
    Elem[result, e, esize] = Elem[result, e, esize] + res;
V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

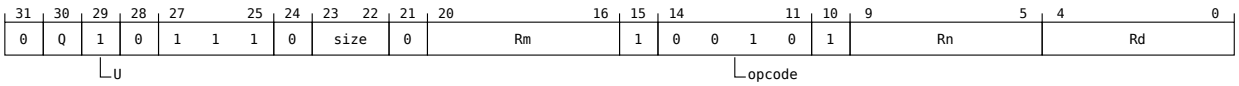
C7.2.343 UDOT (vector)

Dot product unsigned arithmetic (vector)

This instruction performs the dot product of the four unsigned 8-bit elements in each 32-bit element of the first source register with the four unsigned 8-bit elements of the corresponding 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector
(FEAT_DotProd)



Encoding

```
UDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_DotProd) then EndOfDecode(Decode_UNDEF);
if size != '10' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP third source and destination register, encoded in the “Rd” field.

<**Ta**> Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	2S
1	4S

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Tb**> Is an arrangement specifier, encoded in “Q”:

Q	<Tb>
0	8B
1	16B

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

result = V[d, datasize];
for e = 0 to elements-1
    integer res = 0;
    integer element1, element2;
    for i = 0 to 3
        element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
        element2 = UInt(Elem[operand2, 4 * e + i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = Elem[result, e, esize] + res;
V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.344 UHADD

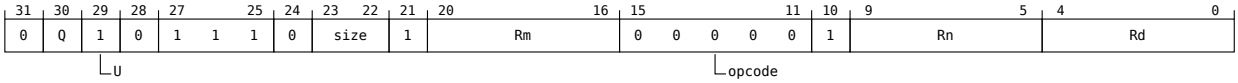
Unsigned halving add

This instruction adds corresponding unsigned integer values from the two source SIMD&FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD&FP register.

The results are truncated. For rounded results, see URHADD.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type
(FEAT_AdvSIMD)



Encoding

```
UHADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    element2 = UInt(Elem[operand2, e, esize]);
    sum = (element1 + element2) >> 1;
    Elem[result, e, esize] = sum<esize-1:0>;

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

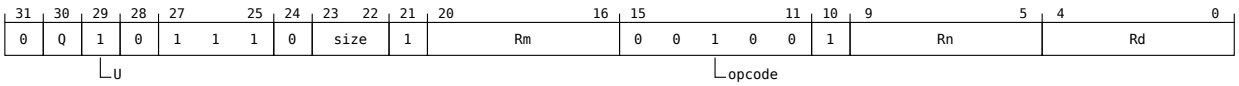
C7.2.345 UHSUB

Unsigned halving subtract

This instruction subtracts the vector elements in the second source SIMD&FP register from the corresponding vector elements in the first source SIMD&FP register, shifts each result right one bit, places each result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type
(FEAT_AdvSIMD)



Encoding

```
UHSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer diff;

for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    element2 = UInt(Elem[operand2, e, esize]);
    diff = (element1 - element2) >> 1;
    Elem[result, e, esize] = diff<esize-1:0>;

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

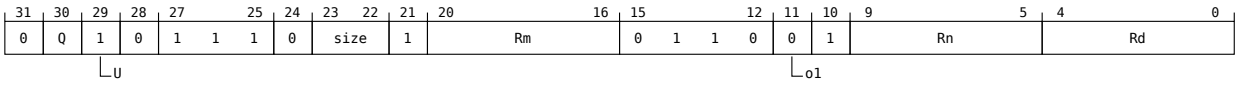
C7.2.346 UMAX

Unsigned maximum (vector)

This instruction compares corresponding elements in the vectors in the two source SIMD&FP registers, places the larger of each pair of unsigned integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type
(FEAT_AdvSIMD)



Encoding

```
UMAX <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer max;

for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    element2 = UInt(Elem[operand2, e, esize]);
    max = Max(element1, element2);
    Elem[result, e, esize] = max<esize-1:0>;

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

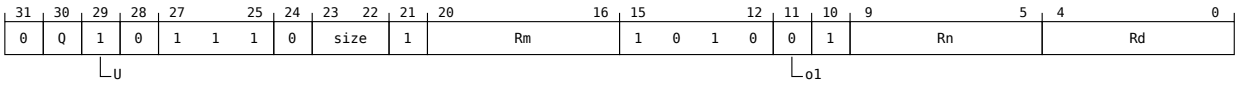
C7.2.347 UMAXP

Unsigned maximum pairwise

This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the largest of each pair of unsigned integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type
(FEAT_AdvSIMD)



Encoding

```
UMAXP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
constant bits(2*datasize) concat = operand2:operand1;
integer element1;
integer element2;
integer max;

for e = 0 to elements-1
    element1 = UInt(Elem[concat, 2*e, esize]);
    element2 = UInt(Elem[concat, (2*e)+1, esize]);
    max = Max(element1, element2);
    Elem[result, e, esize] = max<esize-1:0>;

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

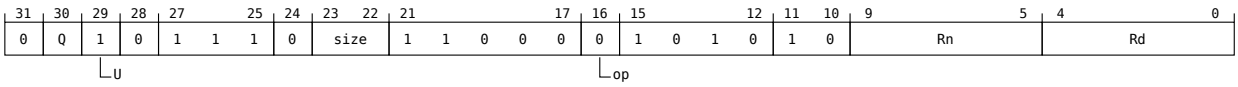
C7.2.348 UMAXV

Unsigned maximum across vector

This instruction compares all the vector elements in the source SIMD&FP register, and writes the largest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Advanced SIMD
(FEAT_AdvSIMD)



Encoding

```
UMAXV <V><d>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '100' then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;

constant boolean unsigned = TRUE;
```

Assembler Symbols

<**V**> Is the destination width specifier, encoded in "size":

size	<V>
00	B
01	H
10	S
11	RESERVED

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<**T**> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	Q	<T>
00	1	16B
01	0	4H
01	1	8H
10	0	RESERVED
10	1	4S
11	x	RESERVED

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(esize) opelt = Elem[operand, 0, esize];
integer max = if unsigned then UInt(opelt) else SInt(opelt);

for e = 1 to elements-1
    opelt = Elem[operand, e, esize];
    constant integer element = if unsigned then UInt(opelt) else SInt(opelt);
    max = Max(max, element);

V[d, esize] = max<esize-1:0>;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

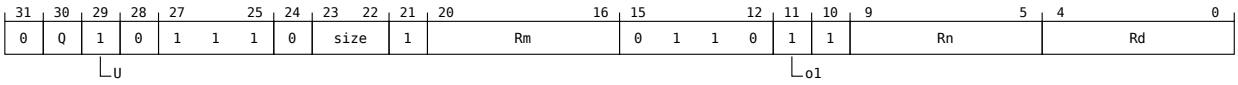
C7.2.349 UMIN

Unsigned minimum (vector)

This instruction compares corresponding vector elements in the two source SIMD&FP registers, places the smaller of each of the two unsigned integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type
(FEAT_AdvSIMD)



Encoding

```
UMIN <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer min;

for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    element2 = UInt(Elem[operand2, e, esize]);
    min = Min(element1, element2);
    Elem[result, e, esize] = min<esize-1:0>;

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

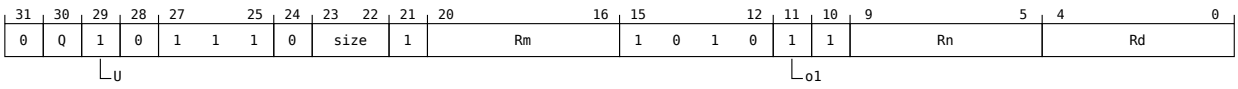
C7.2.350 UMINP

Unsigned minimum pairwise

This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the smallest of each pair of unsigned integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type
(FEAT_AdvSIMD)



Encoding

```
UMINP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
constant bits(2*datasize) concat = operand2:operand1;
integer element1;
integer element2;
integer min;

for e = 0 to elements-1
    element1 = UInt(Elem[concat, 2*e, esize]);
    element2 = UInt(Elem[concat, (2*e)+1, esize]);
    min = Min(element1, element2);
    Elem[result, e, esize] = min<esize-1:0>;

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

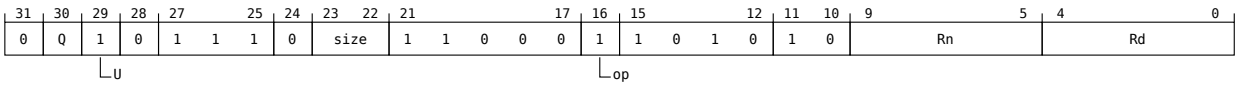
C7.2.351 UMINV

Unsigned minimum across vector

This instruction compares all the vector elements in the source SIMD&FP register, and writes the smallest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Advanced SIMD
(FEAT_AdvSIMD)



Encoding

```
UMINV <V><d>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '100' then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;

constant boolean unsigned = TRUE;
```

Assembler Symbols

<**V**> Is the destination width specifier, encoded in "size":

size	<V>
00	B
01	H
10	S
11	RESERVED

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<**T**> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	Q	<T>
00	1	16B
01	0	4H
01	1	8H
10	0	RESERVED
10	1	4S
11	x	RESERVED

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(esize) opelt = Elem[operand, 0, esize];
integer min = if unsigned then UInt(opelt) else SInt(opelt);

for e = 1 to elements-1
    opelt = Elem[operand, e, esize];
    constant integer element = if unsigned then UInt(opelt) else SInt(opelt);
    min = Min(min, element);

V[d, esize] = min<esize-1:0>;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.352 UMLAL, UMLAL2 (by element)

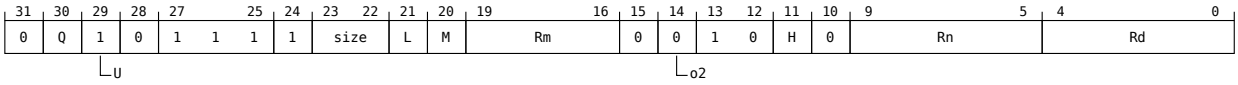
Unsigned multiply-add long (vector, by element)

This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The `UMLAL` instruction extracts vector elements from the lower half of the first source register. The `UMLAL2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector
(FEAT_AdvSIMD)



Encoding

```
UMLAL{2} <Vd>.<Ta>, <Vn>.<Tb>, V<m>.<Ts>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer idxdsize = 64 << UInt(H);
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rmhi:Rm);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Ta**> Is an arrangement specifier, encoded in “size”:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Tb**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**m**> Is the number of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<m>
00	RESERVED
01	UInt('0':Rm)
10	UInt(M:Rm)
11	RESERVED

Restricted to 0-15 when element size <Ts> is H.

<**Ts**> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<**index**> Is the element index, encoded in “size:H:L:M”:

size	<index>
00	RESERVED
01	UInt(H:L:M)
10	UInt(H:L)
11	RESERVED

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize)   operand1 = Vpart[n, part, datasize];
constant bits(idxdsize)   operand2 = V[m, idxdsize];
constant bits(2*datasize) operand3 = V[d, 2*datasize];
bits(2*datasize) result;
integer element1;
constant integer element2 = UInt(Elem[operand2, index, esize]);
bits(2*esize) product;

for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    product = (element1 * element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] + product;

V[d, 2*datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.353 UMLAL, UMLAL2 (vector)

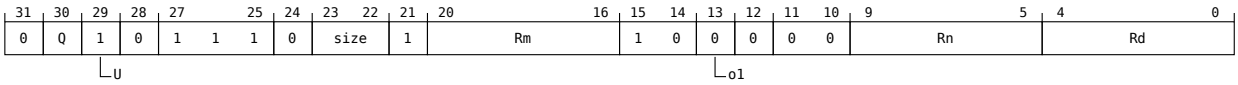
Unsigned multiply-add long (vector)

This instruction multiplies the vector elements in the lower or upper half of the first source SIMD&FP register by the corresponding vector elements of the second source SIMD&FP register, and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The `UMLAL` instruction extracts vector elements from the lower half of the first source register. The `UMLAL2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers, not all the same type
(FEAT_AdvSIMD)



Encoding

```
UMLAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "size":

size	<Ta>
00	8H

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<Ta>
01	4S
10	2D
11	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Tb**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = Vpart[n, part, datasize];
constant bits(datasize) operand2 = Vpart[m, part, datasize];
constant bits(2*datasize) operand3 = V[d, 2*datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
bits(2*esize) accum;

for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    element2 = UInt(Elem[operand2, e, esize]);
    product = (element1 * element2)<2*esize-1:0>;
    accum = Elem[operand3, e, 2*esize] + product;
    Elem[result, e, 2*esize] = accum;

V[d, 2*datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.354 UMLSL, UMLSL2 (by element)

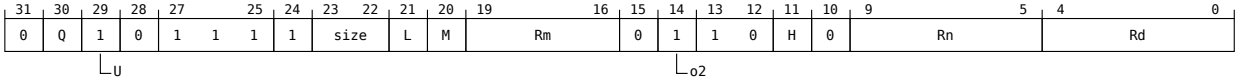
Unsigned multiply-subtract long (vector, by element)

This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register and subtracts the results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The `UMLSL` instruction extracts vector elements from the lower half of the first source register. The `UMLSL2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector
(FEAT_AdvSIMD)



Encoding

```
UMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, V<m>.<Ts>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer idxdsize = 64 << UInt(H);
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rmhi:Rm);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Ta**> Is an arrangement specifier, encoded in “size”:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Tb**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**m**> Is the number of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<m>
00	RESERVED
01	UInt('0':Rm)
10	UInt(M:Rm)
11	RESERVED

Restricted to 0-15 when element size <Ts> is H.

<**Ts**> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<**index**> Is the element index, encoded in “size:H:L:M”:

size	<index>
00	RESERVED
01	UInt(H:L:M)
10	UInt(H:L)
11	RESERVED

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize)   operand1 = Vpart[n, part, datasize];
constant bits(idxdsize)   operand2 = V[m, idxdsize];
constant bits(2*datasize) operand3 = V[d, 2*datasize];
bits(2*datasize) result;
integer element1;
constant integer element2 = UInt(Elem[operand2, index, esize]);
bits(2*esize) product;

for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    product = (element1 * element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] - product;

V[d, 2*datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.355 UMLSL, UMLSL2 (vector)

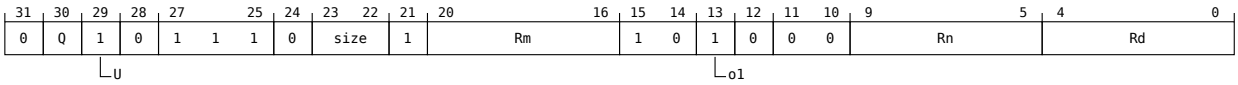
Unsigned multiply-subtract long (vector)

This instruction multiplies corresponding vector elements in the lower or upper half of the two source SIMD&FP registers, and subtracts the results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are unsigned integer values.

The `UMLSL` instruction extracts each source vector from the lower half of each source register. The `UMLSL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers, not all the same type
(FEAT_AdvSIMD)



Encoding

```
UMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "size":

size	<Ta>
00	8H

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<Ta>
01	4S
10	2D
11	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Tb**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = Vpart[n, part, datasize];
constant bits(datasize) operand2 = Vpart[m, part, datasize];
constant bits(2*datasize) operand3 = V[d, 2*datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
bits(2*esize) accum;

for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    element2 = UInt(Elem[operand2, e, esize]);
    product = (element1 * element2)<2*esize-1:0>;
    accum = Elem[operand3, e, 2*esize] - product;
    Elem[result, e, 2*esize] = accum;

V[d, 2*datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.356 UMOV

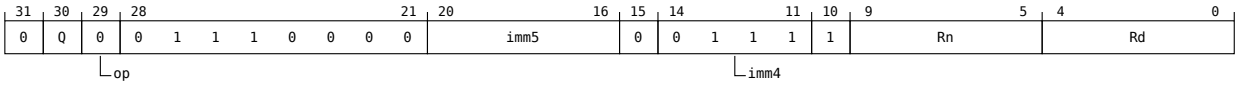
Unsigned move vector element to general-purpose register

This instruction reads the unsigned integer from the source SIMD&FP register, zero-extends it to form a 32-bit or 64-bit value, and writes the result to the destination general-purpose register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias MOV (to general).

Advanced SIMD
(FEAT_AdvSIMD)



Encoding for the 32-bit variant Applies when `(Q == 0)`

```
UMOV <Wd>, <Vn>.<Ts>[<index>]
```

Encoding for the 64-bit variant Applies when `(Q == 1 && imm5 == x1000)`

```
UMOV <Xd>, <Vn>.D[<index>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if imm5 == 'x0000' then EndOfDecode(Decode_UNDEF);
constant integer size = LowestSetBitNZ(imm5<3:0>);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << size;
constant integer datasize = 32 << UInt(Q);
if datasize == 64 && esize < 64 then EndOfDecode(Decode_UNDEF);
if datasize == 32 && esize >= 64 then EndOfDecode(Decode_UNDEF);
constant integer index = UInt(imm5<4:size+1>);
constant integer idxdsize = 64 << UInt(imm5<4>);
```

Assembler Symbols

<**Wd**> Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.

<**Vn**> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<**Ts**> Is an element size specifier, encoded in “imm5”:

imm5	<Ts>
xx000	RESERVED
xxxx1	B
xxx10	H
xx100	S

<**index**> For the “32-bit” variant: is the element index encoded in “imm5”:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

imm5	<index>
xx000	RESERVED
xxxx1	UInt(imm5<4:1>)
xxx10	UInt(imm5<4:2>)
xx100	UInt(imm5<4:3>)

For the “64-bit” variant: is the element index encoded in “imm5<4>”.

<**Xd**> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.

Alias Conditions

Alias	Of variant	Is preferred when
MOV (to general)	32-bit	imm5 IN {'xx100'}
MOV (to general)	64-bit	imm5 IN {'x1000'}

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(idxdsize) operand = V[n, idxdsize];

X[d, datasize] = ZeroExtend(Elem[operand, index, esize], datasize);
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.357 UMULL, UMULL2 (by element)

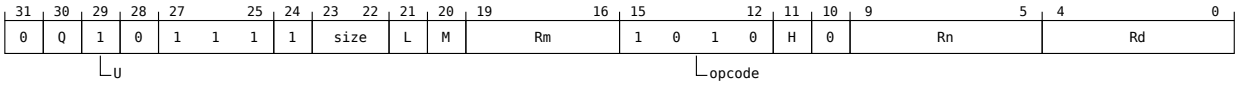
Unsigned multiply long (vector, by element)

This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The `UMULL` instruction extracts vector elements from the lower half of the first source register. The `UMULL2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector
(FEAT_AdvSIMD)



Encoding

```
UMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, V<m>.<Ts>[<index>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer idxdsize = 64 << UInt(H);
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rmhi:Rm);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Ta**> Is an arrangement specifier, encoded in “size”:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Tb**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**m**> Is the number of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<m>
00	RESERVED
01	UInt('0':Rm)
10	UInt(M:Rm)
11	RESERVED

Restricted to 0-15 when element size <Ts> is H.

<**Ts**> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<**index**> Is the element index, encoded in “size:H:L:M”:

size	<index>
00	RESERVED
01	UInt(H:L:M)
10	UInt(H:L)
11	RESERVED

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize)   operand1 = Vpart[n, part, datasize];
constant bits(idxdsize)   operand2 = V[m, idxdsize];
bits(2*datasize) result;
integer element1;
constant integer element2 = UInt(Elem[operand2, index, esize]);
bits(2*esize) product;

for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    product = (element1 * element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = product;

V[d, 2*datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.358 UMULL, UMULL2 (vector)

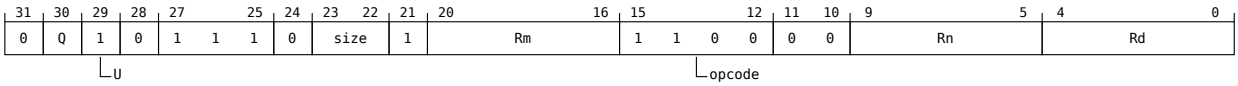
Unsigned multiply long (vector)

This instruction multiplies corresponding vector elements in the lower or upper half of the two source SIMD&FP registers, places the result in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are unsigned integer values.

The `UMULL` instruction extracts each source vector from the lower half of each source register. The `UMULL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers, not all the same type
(FEAT_AdvSIMD)



Encoding

`UMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>`

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "size":

size	<Ta>
00	8H

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<Ta>
01	4S
10	2D
11	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Tb**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize)   operand1 = Vpart[n, part, datasize];
constant bits(datasize)   operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;

for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    element2 = UInt(Elem[operand2, e, esize]);
    Elem[result, e, 2*esize] = (element1 * element2)<2*esize-1:0>;

V[d, 2*datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.359 UQADD

Unsigned saturating add

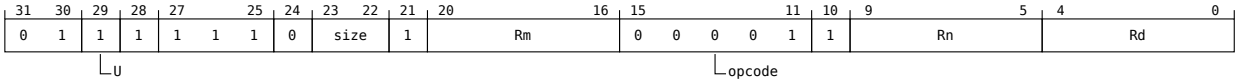
This instruction adds the values of corresponding elements of the two source SIMD&FP registers, places the results into a vector, and writes the vector to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



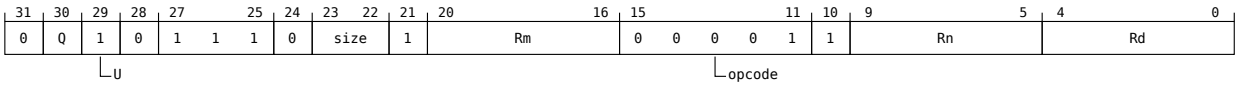
Encoding

```
UQADD <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
constant boolean unsigned = TRUE;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
UQADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
constant boolean unsigned = TRUE;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<**V**> Is a width specifier, encoded in "size":

size	<V>
00	B
01	H
10	S
11	D

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**n**> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<**m**> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer sum;
boolean sat;

for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    element2 = UInt(Elem[operand2, e, esize]);
    sum = element1 + element2;
    (Elem[result, e, esize], sat) = SatQ(sum, esize, unsigned);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.360 UQRSHL

Unsigned saturating rounding shift left (register)

This instruction takes each vector element of the first source SIMD&FP register, shifts the vector element by a value from the least significant byte of the corresponding vector element of the second source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

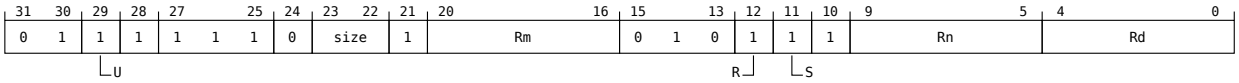
If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are rounded. For truncated results, see UQSHL.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



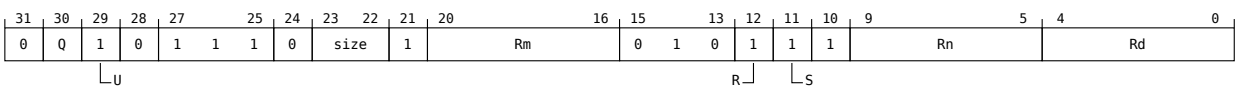
Encoding

```
UQRSHL <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if S == '0' && size != '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant boolean unsigned = TRUE;
constant boolean rounding = TRUE;
constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
UQRSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant boolean unsigned = TRUE;
constant boolean rounding = TRUE;
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**V**> Is a width specifier, encoded in "size":

size	<V>
00	B
01	H
10	S
11	D

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**n**> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<**m**> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
boolean sat;

for e = 0 to elements-1
    integer element = UInt(Elem[operand1, e, esize]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    integer shift = ShiftSat(SInt(Elem[operand2, e, esize]<7:0>), esize);
    if shift >= 0 then // left shift
        element = element << shift;
    else               // right shift
        shift = -shift;
        element = RShr(element, shift, rounding);

    (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.361 UQRSHRN, UQRSHRN2

Unsigned saturating rounded shift right narrow (immediate)

This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are rounded. For truncated results, see UQSHRN.

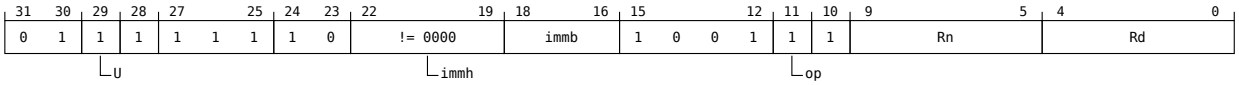
The `UQRSHRN` instruction writes the vector to the lower half of the destination register and clears the upper half. The `UQRSHRN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



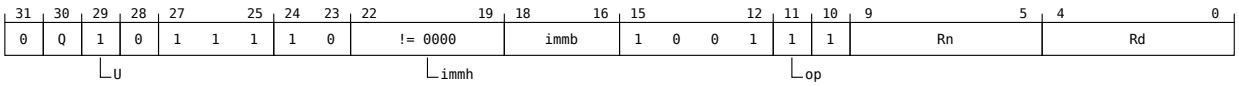
Encoding

```
UQRSHRN <Vb><d>, <Va><n>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh == '0000' then EndOfDecode(Decode_UNDEF);
if immh<3> == '1' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << HighestSetBitNZ(immh<2:0>);
constant integer datasize = esize;
constant integer elements = 1;
constant integer part = 0;

constant integer shift = (2 * esize) - UInt(immh:immb);
constant boolean round = TRUE;
constant boolean unsigned = TRUE;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
UQRSHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh<3> == '1' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << HighestSetBitNZ(immh<2:0>);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;

constant integer shift = (2 * esize) - UInt(immh:immb);
constant boolean round = TRUE;
constant boolean unsigned = TRUE;
```

Assembler Symbols

<**Vb**> Is the destination width specifier, encoded in “immh”:

immh	<**Vb**>
0001	B
001x	H
01xx	S
1xxx	RESERVED

<**d**> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<**Va**> Is the source width specifier, encoded in “immh”:

immh	<**Va**>
0001	H
001x	S
01xx	D
1xxx	RESERVED

<**n**> Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<**shift**> For the “Scalar” variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in “immh:immb”:

immh	<**shift**>
0001	16 - UInt(immh:immb)
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	RESERVED

For the “Vector” variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immb”:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

immh	<shift>
0001	16 - UInt(immh:immb)
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	RESERVED

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

Q	2
0	[absent]
1	[present]

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**Tb**> Is an arrangement specifier, encoded in "immh:Q":

immh	Q	<Tb>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<**Ta**> Is an arrangement specifier, encoded in "immh":

immh	<Ta>
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize*2) operand = V[n, datasize*2];
bits(datasize) result;
boolean sat;

for e = 0 to elements-1
    constant bits(2*esize) opelt = Elem[operand, e, 2*esize];
    if unsigned then
        constant integer element = RShr(UInt(opelt), shift, round);
        (Elem[result, e, esize], sat) = UnsignedSatQ(element, esize);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        constant integer element = RShr(SInt(opelt), shift, round);
        (Elem[result, e, esize], sat) = SignedSatQ(element, esize);
    if sat then FPSR.QC = '1';

Vpart[d, part, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.362 UQSHL (immediate)

Unsigned saturating shift left (immediate)

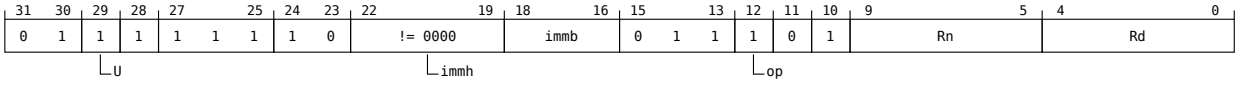
This instruction takes each vector element in the source SIMD&FP register, shifts it by an immediate value, places the results in a vector, and writes the vector to the destination SIMD&FP register. The results are truncated. For rounded results, see UQRSHL.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



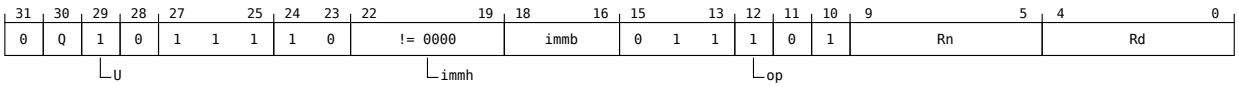
Encoding

```
UQSHL <V><d>, <V><n>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh == '0000' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << HighestSetBitNZ(immh);
constant integer datasize = esize;
constant integer elements = 1;
constant integer shift = UInt(immh:immb) - esize;

constant boolean src_unsigned = TRUE;
constant boolean dst_unsigned = TRUE;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
UQSHL <Vd>.<T>, <Vn>.<T>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh<3>:Q == '10' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer n = UInt(Rn);
constant integer esize = 8 << HighestSetBitNZ(immh);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
constant integer shift = UInt(immh:immb) - esize;

constant boolean src_unsigned = TRUE;
constant boolean dst_unsigned = TRUE;
```

Assembler Symbols

<**V**> Is a width specifier, encoded in “immh”:

immh	<V>
0001	B
001x	H
01xx	S
1xxx	D

<**d**> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<**n**> Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<**shift**> For the “Scalar” variant: is the left shift amount, in the range 0 to the operand width in bits minus 1, encoded in “immh:immb”:

immh	<shift>
0001	UInt(immh:immb) - 8
001x	UInt(immh:immb) - 16
01xx	UInt(immh:immb) - 32
1xxx	UInt(immh:immb) - 64

For the “Vector” variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in “immh:immb”:

immh	<shift>
0001	UInt(immh:immb) - 8
001x	UInt(immh:immb) - 16
01xx	UInt(immh:immb) - 32
1xxx	UInt(immh:immb) - 64

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0001	0	8B
0001	1	16B

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

immh	Q	<T>
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand  = V[n, datasize];
bits(datasize) result;
boolean sat;

for e = 0 to elements-1
    constant bits(esize) opelt = Elem[operand, e, esize];
    constant integer element = if src_unsigned then UInt(opelt) else SInt(opelt);
    (Elem[result, e, esize], sat) = SatQ(element << shift, esize, dst_unsigned);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.363 UQSHL (register)

Unsigned saturating shift left (register)

This instruction takes each element in the vector of the first source SIMD&FP register, shifts the element by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

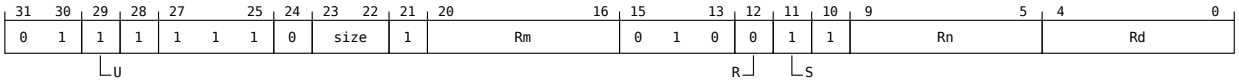
If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are truncated. For rounded results, see UQRSHL.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



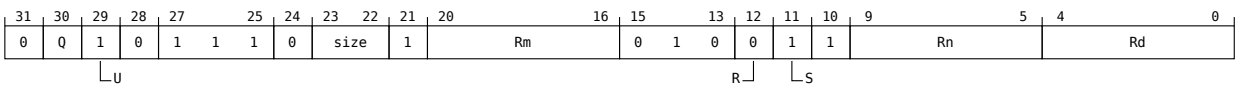
Encoding

```
UQSHL <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if S == '0' && size != '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant boolean unsigned = TRUE;
constant boolean rounding = FALSE;
constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
UQSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant boolean unsigned = TRUE;
constant boolean rounding = FALSE;
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**V**> Is a width specifier, encoded in "size":

size	<V>
00	B
01	H
10	S
11	D

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**n**> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<**m**> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
boolean sat;

for e = 0 to elements-1
    integer element = UInt(Elem[operand1, e, esize]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    integer shift = ShiftSat(SInt(Elem[operand2, e, esize]<7:0>), esize);
    if shift >= 0 then // left shift
        element = element << shift;
    else               // right shift
        shift = -shift;
        element = RShr(element, shift, rounding);

    (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.364 UQSHRN, UQSHRN2

Unsigned saturating shift right narrow (immediate)

This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, saturates each shifted result to a value that is half the original width, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are truncated. For rounded results, see UQRSHRN.

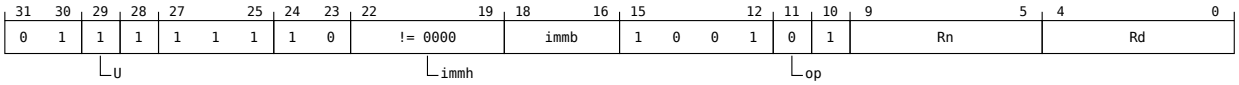
The `UQSHRN` instruction writes the vector to the lower half of the destination register and clears the upper half. The `UQSHRN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



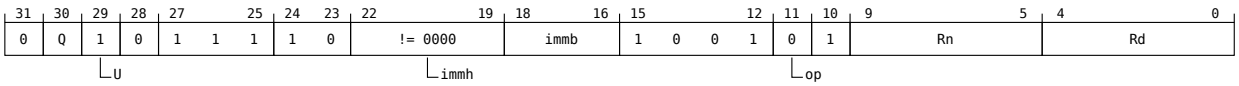
Encoding

```
UQSHRN <Vb><d>, <Va><n>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh == '0000' then EndOfDecode(Decode_UNDEF);
if immh<3> == '1' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << HighestSetBitNZ(immh<2:0>);
constant integer datasize = esize;
constant integer elements = 1;
constant integer part = 0;

constant integer shift = (2 * esize) - UInt(immh:immb);
constant boolean round = FALSE;
constant boolean unsigned = TRUE;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
UQSHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh<3> == '1' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << HighestSetBitNZ(immh<2:0>);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;

constant integer shift = (2 * esize) - UInt(immh:immb);
constant boolean round = FALSE;
constant boolean unsigned = TRUE;
```

Assembler Symbols

<**Vb**> Is the destination width specifier, encoded in “immh”:

immh	<**Vb**>
0001	B
001x	H
01xx	S
1xxx	RESERVED

<**d**> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<**Va**> Is the source width specifier, encoded in “immh”:

immh	<**Va**>
0001	H
001x	S
01xx	D
1xxx	RESERVED

<**n**> Is the number of the SIMD&FP source register, encoded in the “Rn” field.

<**shift**> For the “Scalar” variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in “immh:immb”:

immh	<**shift**>
0001	16 - UInt(immh:immb)
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	RESERVED

For the “Vector” variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immb”:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

immh	<shift>
0001	16 - UInt(immh:immb)
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	RESERVED

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

Q	2
0	[absent]
1	[present]

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**Tb**> Is an arrangement specifier, encoded in "immh:Q":

immh	Q	<Tb>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<**Ta**> Is an arrangement specifier, encoded in "immh":

immh	<Ta>
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize*2) operand = V[n, datasize*2];
bits(datasize) result;
boolean sat;

for e = 0 to elements-1
    constant bits(2*esize) opelt = Elem[operand, e, 2*esize];
    if unsigned then
        constant integer element = RShr(UInt(opelt), shift, round);
        (Elem[result, e, esize], sat) = UnsignedSatQ(element, esize);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            constant integer element = RShr(SInt(opelt), shift, round);
            (Elem[result, e, esize], sat) = SignedSatQ(element, esize);
        if sat then FPSR.QC = '1';

    Vpart[d, part, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.365 UQSUB

Unsigned saturating subtract

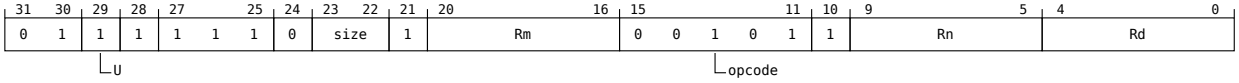
This instruction subtracts the element values of the second source SIMD&FP register from the corresponding element values of the first source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



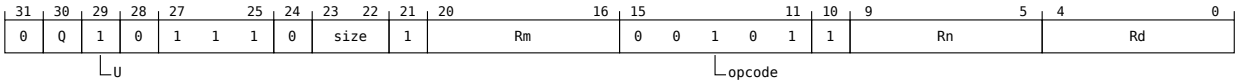
Encoding

```
UQSUB <V><d>, <V><n>, <V><m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
constant boolean unsigned = TRUE;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
UQSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
constant boolean unsigned = TRUE;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<**V**> Is a width specifier, encoded in "size":

size	<V>
00	B
01	H
10	S
11	D

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**n**> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<**m**> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer diff;
boolean sat;

for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    element2 = UInt(Elem[operand2, e, esize]);
    diff = element1 - element2;
    (Elem[result, e, esize], sat) = SatQ(diff, esize, unsigned);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.366 UQXTN, UQXTN2

Unsigned saturating extract narrow

This instruction reads each vector element from the source SIMD&FP register, saturates each value to half the original width, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

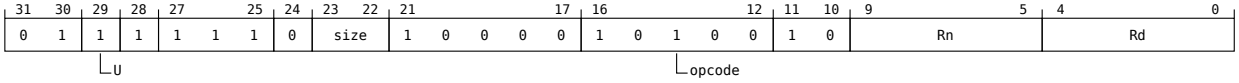
If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The `UQXTN` instruction writes the vector to the lower half of the destination register and clears the upper half. The `UQXTN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



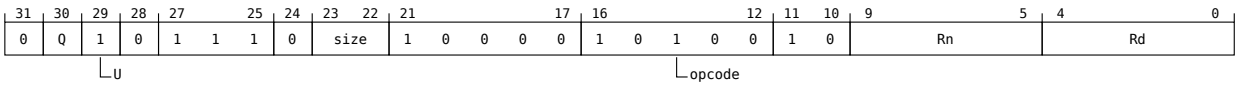
Encoding

```
UQXTN <Vb><d>, <Va><n>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer part = 0;
constant integer elements = 1;
constant boolean unsigned = TRUE;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
UQXTN{2} <Vd>.<Tb>, <Vn>.<Ta>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer esize = 8 << UInt(size);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;
constant boolean unsigned = TRUE;
```

Assembler Symbols

<**Vb**> Is the destination width specifier, encoded in "size":

size	<Vb>
00	B
01	H
10	S
11	RESERVED

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**Va**> Is the source width specifier, encoded in "size":

size	<Va>
00	H
01	S
10	D
11	RESERVED

<**n**> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

Q	2
0	[absent]
1	[present]

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**Tb**> Is an arrangement specifier, encoded in "size:Q":

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	Q	<Tb>
11	x	RESERVED

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<**Ta**> Is an arrangement specifier, encoded in "size":

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(2*datasize) operand = V[n, 2*datasize];
bits(datasize) result;
bits(2*esize) element;
boolean sat;

for e = 0 to elements-1
    element = Elem[operand, e, 2*esize];
    (Elem[result, e, esize], sat) = SatQ(UInt(element), esize, unsigned);
    if sat then FPSR.QC = '1';

Vpart[d, part, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

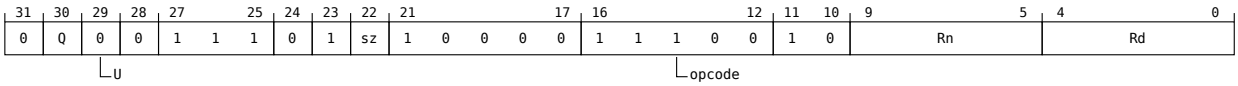
C7.2.367 URECPE

Unsigned reciprocal estimate

This instruction reads each vector element from the source SIMD&FP register, calculates an approximate inverse for the unsigned integer value, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector
(FEAT_AdvSIMD)



Encoding

```
URECPE <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

if sz == '1' then EndOfDecode(Decode_UNDEF);
constant integer esize = 32;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T> Is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(32) element;

for e = 0 to elements-1
    element = Elem[operand, e, 32];
    Elem[result, e, 32] = UnsignedRecipEstimate(element);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.368 URHADD

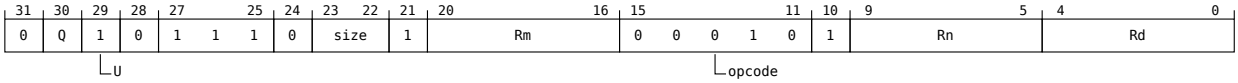
Unsigned rounding halving add

This instruction adds corresponding unsigned integer values from the two source SIMD&FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD&FP register.

The results are rounded. For truncated results, see UHADD.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type
(FEAT_AdvSIMD)



Encoding

```
URHADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    element2 = UInt(Elem[operand2, e, esize]);
    sum = (element1 + element2 + 1) >> 1;
    Elem[result, e, esize] = sum<esize-1:0>;

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.369 URSHL

Unsigned rounding shift left (register)

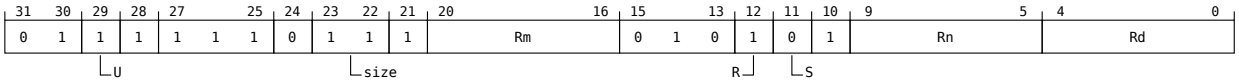
This instruction takes each element in the vector of the first source SIMD&FP register, shifts the vector element by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a rounding right shift.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



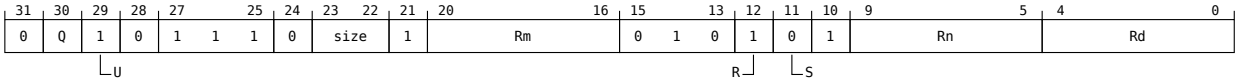
Encoding

```
URSHL D<d>, D<n>, D<m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if S == '0' && size != '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant boolean rounding = TRUE;
constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
URSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant boolean rounding = TRUE;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**d**> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<**n**> Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<**m**> Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

for e = 0 to elements-1
    integer element = UInt(Elem[operand1, e, esize]);
    integer shift = ShiftSat(SInt(Elem[operand2, e, esize]<7:0>), esize);
    if shift >= 0 then // left shift
        element = element << shift;
    else               // right shift
        shift = -shift;
        element = RShr(element, shift, rounding);

    Elem[result, e, esize] = element<esize-1:0>;

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.370 URSHR

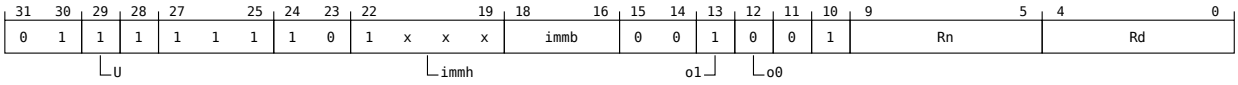
Unsigned rounding shift right (immediate)

This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are rounded. For truncated results, see USHR.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



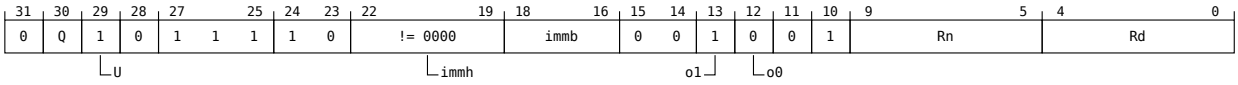
Encoding

```
URSHR D<d>, D<n>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh<3> != '1' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << 3;
constant integer datasize = esize;
constant integer elements = 1;

constant integer shift = (esize * 2) - UInt(immh:immb);
constant boolean unsigned = TRUE;
constant boolean round = TRUE;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
URSHR <Vd>.<T>, <Vn>.<T>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh<3>:Q == '10' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << HighestSetBitNZ(immh);
constant integer datasize = 64 << UInt(Q);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer elements = datasize DIV esize;

constant integer shift = (esize * 2) - UInt(immh:immb);
constant boolean unsigned = TRUE;
constant boolean round = TRUE;
```

Assembler Symbols

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**n**> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<**shift**> For the "Scalar" variant: is the right shift amount, in the range 1 to 64, encoded as 128 - UInt("immh:immb").

For the "Vector" variant: is the right shift amount, in the range 1 to the element width in bits, encoded in "immh:immb":

immh	<shift>
0001	16 - UInt(immh:immb)
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	128 - UInt(immh:immb)

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> Is an arrangement specifier, encoded in "immh:Q":

immh	Q	<T>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand  = V[n, datasize];
bits(datasize) result;
integer element;

for e = 0 to elements-1
    constant bits(esize) opelt = Elem[operand, e, esize];
    if unsigned then
        element = RShr(UInt(opelt), shift, round);
    else
        element = RShr(SInt(opelt), shift, round);
    Elem[result, e, esize] = element<esize-1:0>;

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

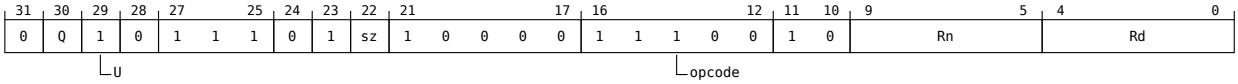
C7.2.371 URSQRTE

Unsigned reciprocal square root estimate

This instruction reads each vector element from the source SIMD&FP register, calculates an approximate inverse square root for each value, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector
(FEAT_AdvSIMD)



Encoding

```
URSQRTE <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

if sz == '1' then EndOfDecode(Decode_UNDEF);
constant integer esize = 32;
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> Is an arrangement specifier, encoded in "sz:Q":

sz	Q	<T>
0	0	2S
0	1	4S
1	x	RESERVED

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(32) element;

for e = 0 to elements-1
    element = Elem[operand, e, 32];
    Elem[result, e, 32] = UnsignedRSqrtEstimate(element);

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.372 URSRA

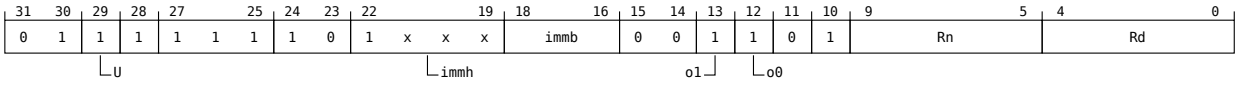
Unsigned rounding shift right and accumulate (immediate)

This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are rounded. For truncated results, see USRA.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



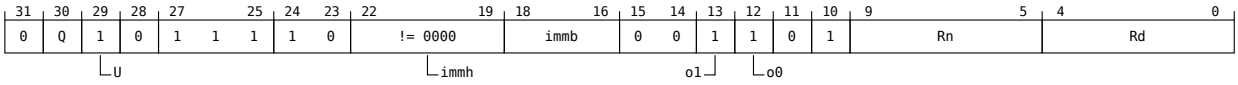
Encoding

```
URSRA D<d>, D<n>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh<3> != '1' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << 3;
constant integer datasize = esize;
constant integer elements = 1;

constant integer shift = (esize * 2) - UInt(immh:immb);
constant boolean unsigned = TRUE;
constant boolean round = TRUE;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
URSRA <Vd>.<T>, <Vn>.<T>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh<3>:Q == '10' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << HighestSetBitNZ(immh);
constant integer datasize = 64 << UInt(Q);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer elements = datasize DIV esize;

constant integer shift = (esize * 2) - UInt(immh:immb);
constant boolean unsigned = TRUE;
constant boolean round = TRUE;
```

Assembler Symbols

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**n**> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<**shift**> For the "Scalar" variant: is the right shift amount, in the range 1 to 64, encoded as 128 - UInt("immh:immb").

For the "Vector" variant: is the right shift amount, in the range 1 to the element width in bits, encoded in "immh:immb":

immh	<shift>
0001	16 - UInt(immh:immb)
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	128 - UInt(immh:immb)

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> Is an arrangement specifier, encoded in "immh:Q":

immh	Q	<T>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand  = V[n, datasize];
constant bits(datasize) operand2 = V[d, datasize];
bits(datasize) result;
integer element;

for e = 0 to elements-1
    constant bits(esize) opelt = Elem[operand, e, esize];
    if unsigned then
        element = RShr(UInt(opelt), shift, round);
    else
        element = RShr(SInt(opelt), shift, round);
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.373 USHL

Unsigned shift left (register)

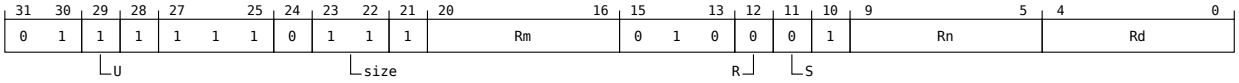
This instruction takes each element in the vector of the first source SIMD&FP register, shifts each element by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a truncating right shift. For a rounding shift, see URSHL.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



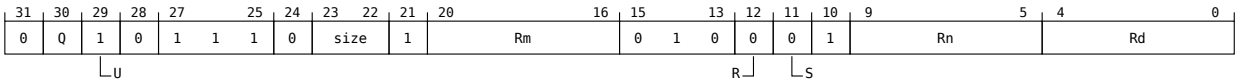
Encoding

```
USHL D<d>, D<n>, D<m>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if S == '0' && size != '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant boolean rounding = FALSE;
constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
USHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant boolean rounding = FALSE;
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

<**d**> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<**n**> Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<**m**> Is the number of the second SIMD&FP source register, encoded in the “Rm” field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

for e = 0 to elements-1
    integer element = UInt(Elem[operand1, e, esize]);
    integer shift = ShiftSat(SInt(Elem[operand2, e, esize]<7:0>), esize);
    if shift >= 0 then // left shift
        element = element << shift;
    else               // right shift
        shift = -shift;
        element = RShr(element, shift, rounding);

    Elem[result, e, esize] = element<esize-1:0>;

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.374 USHLL, USHLL2

Unsigned shift left long (immediate)

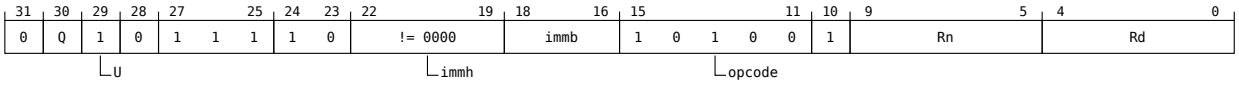
This instruction reads each vector element in the lower or upper half of the source SIMD&FP register, shifts the unsigned integer value left by the specified number of bits, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The `USHLL` instruction extracts vector elements from the lower half of the source register. The `USHLL2` instruction extracts vector elements from the upper half of the source register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias UXTL, UXTL2.

Vector
(FEAT_AdvSIMD)



Encoding

```
USHLL{2} <Vd>.<Ta>, <Vn>.<Tb>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh<3> == '1' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << HighestSetBitNZ(immh<2:0>);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;

constant integer shift = UInt(immh:immb) - esize;
constant boolean unsigned = TRUE;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

Q	2
0	[absent]
1	[present]

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**Ta**> Is an arrangement specifier, encoded in "immh":

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

immh	<Ta>
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<**Tb**> Is an arrangement specifier, encoded in "immh:Q":

immh	Q	<Tb>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<**shift**> Is the left shift amount, in the range 0 to the source element width in bits minus 1, encoded in "immh:immb":

immh	<shift>
0001	UInt(immh:immb) - 8
001x	UInt(immh:immb) - 16
01xx	UInt(immh:immb) - 32
1xxx	RESERVED

Alias Conditions

Alias	Is preferred when
UXTL, UXTL2	immb == '000' &&BitCount(immh) == 1

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = Vpart[n, part, datasize];
bits(datasize*2) result;

for e = 0 to elements-1
    constant bits(esize) opelt = Elem[operand, e, esize];
    constant integer element = if unsigned then UInt(opelt) else SInt(opelt);
    Elem[result, e, 2*esize] = (element << shift)<2*esize-1:0>;

V[d, datasize*2] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.375 USHR

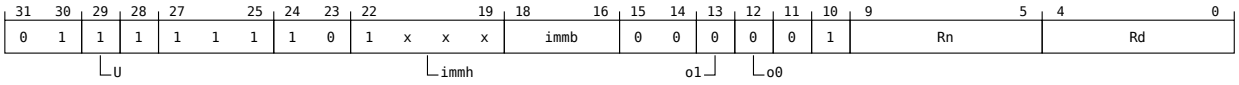
Unsigned shift right (immediate)

This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are truncated. For rounded results, see URSHR.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



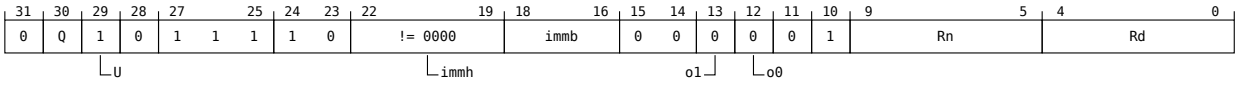
Encoding

```
USHR D<d>, D<n>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh<3> != '1' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << 3;
constant integer datasize = esize;
constant integer elements = 1;

constant integer shift = (esize * 2) - UInt(immh:immb);
constant boolean unsigned = TRUE;
constant boolean round = FALSE;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
USHR <Vd>.<T>, <Vn>.<T>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh<3>:Q == '10' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << HighestSetBitNZ(immh);
constant integer datasize = 64 << UInt(Q);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer elements = datasize DIV esize;

constant integer shift = (esize * 2) - UInt(immh:immb);
constant boolean unsigned = TRUE;
constant boolean round = FALSE;
```

Assembler Symbols

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**n**> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<**shift**> For the "Scalar" variant: is the right shift amount, in the range 1 to 64, encoded as 128 - UInt("immh:immb").

For the "Vector" variant: is the right shift amount, in the range 1 to the element width in bits, encoded in "immh:immb":

immh	<shift>
0001	16 - UInt(immh:immb)
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	128 - UInt(immh:immb)

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> Is an arrangement specifier, encoded in "immh:Q":

immh	Q	<T>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand  = V[n, datasize];
bits(datasize) result;
integer element;

for e = 0 to elements-1
    constant bits(esize) opelt = Elem[operand, e, esize];
    if unsigned then
        element = RShr(UInt(opelt), shift, round);
    else
        element = RShr(SInt(opelt), shift, round);
    Elem[result, e, esize] = element<esize-1:0>;

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.376 USQADD

Unsigned saturating accumulate of signed value

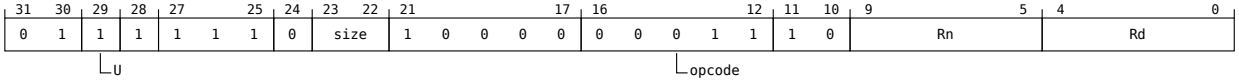
This instruction adds the signed integer values of the vector elements in the source SIMD&FP register to corresponding unsigned integer values of the vector elements in the destination SIMD&FP register, and accumulates the resulting unsigned integer values with the vector elements of the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



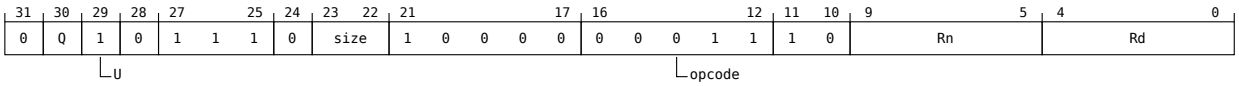
Encoding

```
USQADD <V><d>, <V><n>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << UInt(size);
constant integer datasize = esize;
constant integer elements = 1;
constant boolean unsigned = TRUE;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
USQADD <Vd>.<T>, <Vn>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
constant boolean unsigned = TRUE;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Assembler Symbols

<**V**> Is a width specifier, encoded in "size":

size	<V>
00	B
01	H
10	S
11	D

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**n**> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand = V[n, datasize];
bits(datasize) result;

constant bits(datasize) operand2 = V[d, datasize];
integer op1;
integer op2;
boolean sat;

for e = 0 to elements-1
    op1 = SInt(Elem[operand, e, esize]);
    op2 = UInt(Elem[operand2, e, esize]);
    (Elem[result, e, esize], sat) = SatQ(op1 + op2, esize, unsigned);
    if sat then FPSR.QC = '1';
V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.377 USRA

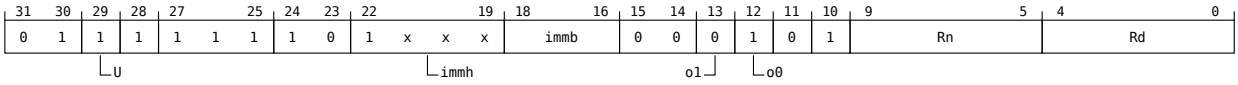
Unsigned shift right and accumulate (immediate)

This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are truncated. For rounded results, see URSRA.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(FEAT_AdvSIMD)



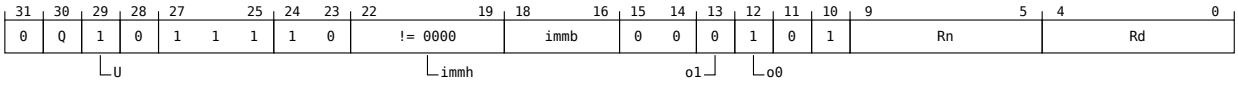
Encoding

```
USRA D<d>, D<n>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh<3> != '1' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << 3;
constant integer datasize = esize;
constant integer elements = 1;

constant integer shift = (esize * 2) - UInt(immh:immb);
constant boolean unsigned = TRUE;
constant boolean round = FALSE;
```

Vector
(FEAT_AdvSIMD)



Encoding

```
USRA <Vd>.<T>, <Vn>.<T>, #<shift>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if immh<3>:Q == '10' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer esize = 8 << HighestSetBitNZ(immh);
constant integer datasize = 64 << UInt(Q);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant integer elements = datasize DIV esize;

constant integer shift = (esize * 2) - UInt(immh:immb);
constant boolean unsigned = TRUE;
constant boolean round = FALSE;
```

Assembler Symbols

<**d**> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<**n**> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<**shift**> For the "Scalar" variant: is the right shift amount, in the range 1 to 64, encoded as 128 - UInt("immh:immb").

For the "Vector" variant: is the right shift amount, in the range 1 to the element width in bits, encoded in "immh:immb":

immh	<shift>
0001	16 - UInt(immh:immb)
001x	32 - UInt(immh:immb)
01xx	64 - UInt(immh:immb)
1xxx	128 - UInt(immh:immb)

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**T**> Is an arrangement specifier, encoded in "immh:Q":

immh	Q	<T>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand  = V[n, datasize];
constant bits(datasize) operand2 = V[d, datasize];
bits(datasize) result;
integer element;

for e = 0 to elements-1
    constant bits(esize) opelt = Elem[operand, e, esize];
    if unsigned then
        element = RShr(UInt(opelt), shift, round);
    else
        element = RShr(SInt(opelt), shift, round);
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

V[d, datasize] = result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.378 USUBL, USUBL2

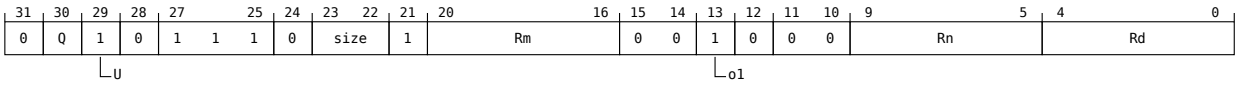
Unsigned subtract long

This instruction subtracts each vector element in the lower or upper half of the second source SIMD&FP register from the corresponding vector element of the first source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The destination vector elements are twice as long as the source vector elements.

The `USUBL` instruction extracts each source vector from the lower half of each source register. The `USUBL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers, not all the same type
(FEAT_AdvSIMD)



Encoding

```
USUBL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "size":

size	<Ta>
00	8H

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<Ta>
01	4S
10	2D
11	RESERVED

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Tb**> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = Vpart[n, part, datasize];
constant bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    element2 = UInt(Elem[operand2, e, esize]);
    sum = element1 - element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d, 2*datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.379 USUBW, USUBW2

Unsigned subtract wide

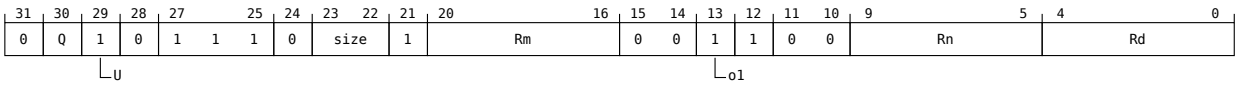
This instruction subtracts each vector element in the lower or upper half of the second source SIMD&FP register from the corresponding vector element in the first source SIMD&FP register, places the result in a vector, and writes the vector to the SIMD&FP destination register. All the values in this instruction are unsigned integer values.

The vector elements of the destination register and the first source register are twice as long as the vector elements of the second source register.

The `USUBW` instruction extracts vector elements from the lower half of the second source register. The `USUBW2` instruction extracts vector elements from the upper half of the second source register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers, not all the same type
(FEAT_AdvSIMD)



Encoding

```
USUBW{2} <Vd>.<Ta>, <Vn>.<Ta>, <Vm>.<Tb>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**Ta**> Is an arrangement specifier, encoded in “size”:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Tb> Is an arrangement specifier, encoded in "size:Q":

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(2*datasize) operand1 = V[n, 2*datasize];
constant bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, 2*esize]);
    element2 = UInt(Elem[operand2, e, esize]);
    sum = element1 - element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d, 2*datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.380 UXTL, UXTL2

Unsigned extend long

This instruction copies each vector element from the lower or upper half of the source SIMD&FP register into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

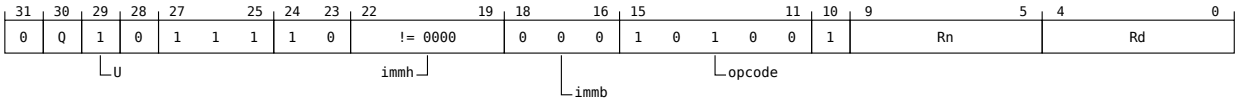
The `UXTL` instruction extracts vector elements from the lower half of the source register. The `UXTL2` instruction extracts vector elements from the upper half of the source register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of USHLL, USHLL2. This means:

- The encodings in this description are named to match the encodings of USHLL, USHLL2.
- The description of USHLL, USHLL2 gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Vector
(FEAT_AdvSIMD)



Encoding

```
UXTL{2} <Vd>.<Ta>, <Vn>.<Tb>
```

is equivalent to

```
USHLL{2} <Vd>.<Ta>, <Vn>.<Tb>, #0
```

and is the preferred disassembly when `BitCount(immh) == 1`.

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

Q	2
0	[absent]
1	[present]

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<**Ta**> Is an arrangement specifier, encoded in "immh":

immh	<Ta>
0001	8H
001x	4S

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

immh	<Ta>
01xx	2D
1xxx	RESERVED

<**Vn**> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<**Tb**> Is an arrangement specifier, encoded in "immh:Q":

immh	Q	<Tb>
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

Operation

The description of USHLL, USHLL2 gives the operational pseudocode for this instruction.

Operational Information

The description of USHLL, USHLL2 gives the operational information for this instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.381 UZP1

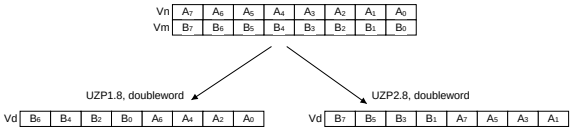
Unzip vectors (primary)

This instruction reads corresponding even-numbered vector elements from the two source SIMD&FP registers, starting at zero, places the result from the first source register into consecutive elements in the lower half of a vector, and the result from the second source register into consecutive elements in the upper half of a vector, and writes the vector to the destination SIMD&FP register.

Note

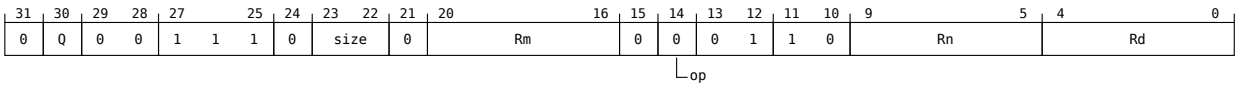
This instruction can be used with `UZP2` to de-interleave two vectors.

The following figure shows an example of the operation of UZP1 and UZP2 with the arrangement specifier 8B.



Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Advanced SIMD
(FEAT_AdvSIMD)



Encoding

```
UZP1 <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
constant integer part = UInt(op);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	Q	<T>
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operandl = V[n, datasize];
constant bits(datasize) operandh = V[m, datasize];
bits(datasize) result;

constant bits(datasize*2) zipped = operandh:operandl;
for e = 0 to elements-1
    Elem[result, e, esize] = Elem[zipped, 2*e+part, esize];

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.382 UZP2

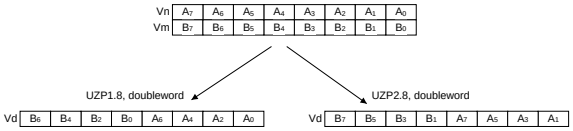
Unzip vectors (secondary)

This instruction reads corresponding odd-numbered vector elements from the two source SIMD&FP registers, places the result from the first source register into consecutive elements in the lower half of a vector, and the result from the second source register into consecutive elements in the upper half of a vector, and writes the vector to the destination SIMD&FP register.

Note

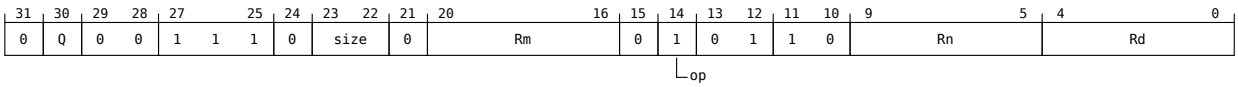
This instruction can be used with `UZP1` to de-interleave two vectors.

The following figure shows an example of the operation of UZP1 and UZP2 with the arrangement specifier 8B.



Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Advanced SIMD
(FEAT_AdvSIMD)



Encoding

```
UZP2 <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
constant integer part = UInt(op);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	0	8B

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	Q	<T>
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operandl = V[n, datasize];
constant bits(datasize) operandh = V[m, datasize];
bits(datasize) result;

constant bits(datasize*2) zipped = operandh:operandl;
for e = 0 to elements-1
    Elem[result, e, esize] = Elem[zipped, 2*e+part, esize];

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

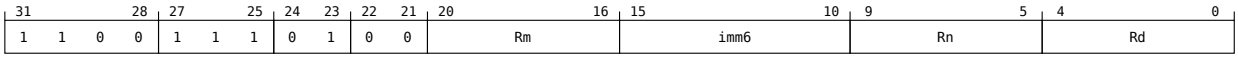
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.383 XAR

Exclusive-OR and rotate

This instruction performs a bitwise exclusive-OR of the 128-bit vectors in the two source SIMD&FP registers, rotates each 64-bit element of the resulting 128-bit vector right by the value specified by a 6-bit immediate value, and writes the result to the destination SIMD&FP register.

Advanced SIMD
(FEAT_SHA3)



Encoding

```
XAR <Vd>.2D, <Vn>.2D, <Vm>.2D, #<imm6>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA3) then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

<imm6> Is a rotation right, encoded in “imm6”.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

constant bits(128) Vm = V[m, 128];
constant bits(128) Vn = V[n, 128];
constant bits(128) tmp = Vn EOR Vm;
V[d, 128] = ROR(tmp<127:64>, UInt(imm6)):ROR(tmp<63:0>, UInt(imm6));
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.384 XTN, XTN2

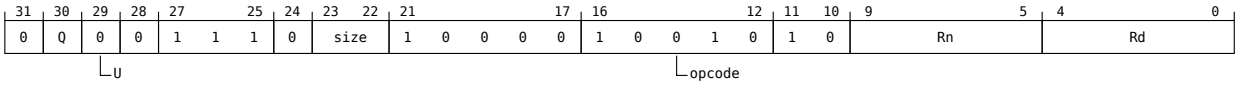
Extract narrow

This instruction reads each vector element from the source SIMD&FP register, narrows each value to half the original width, places the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements.

The XTN instruction writes the vector to the lower half of the destination register and clears the upper half. The XTN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector
(FEAT_AdvSIMD)



Encoding

```
XTN{2} <Vd>.<Tb>, <Vn>.<Ta>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size == '11' then EndOfDecode(Decode_UNDEF);
constant integer d = UInt(Rd);
constant integer n = UInt(Rn);

constant integer esize = 8 << UInt(size);
constant integer datasize = 64;
constant integer part = UInt(Q);
constant integer elements = datasize DIV esize;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in "size:Q":

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	Q	<Tb>
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<**Vn**> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<**Ta**> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(2*datasize) operand = V[n, 2*datasize];
bits(datasize) result;
bits(2*esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, 2*esize];
    Elem[result, e, esize] = element<esize-1:0>;
Vpart[d, part, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.385 ZIP1

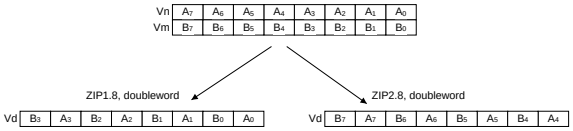
Zip vectors (primary)

This instruction reads adjacent vector elements from the lower half of two source SIMD&FP registers as pairs, interleaves the pairs and places them into a vector, and writes the vector to the destination SIMD&FP register. The first pair from the first source register is placed into the two lowest vector elements, with subsequent pairs taken alternately from each source register.

Note

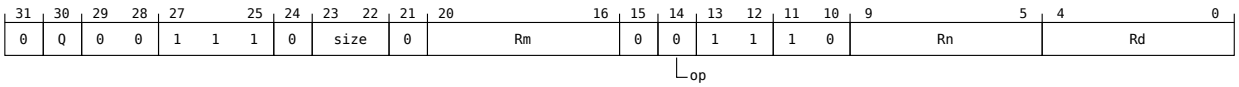
This instruction can be used with `ZIP2` to interleave two vectors.

The following figure shows an example of the operation of ZIP1 and ZIP2 with the arrangement specifier 8B.



Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Advanced SIMD
(FEAT_AdvSIMD)



Encoding

```
ZIP1 <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
constant integer part = UInt(op);
constant integer pairs = elements DIV 2;
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> Is an arrangement specifier, encoded in “size:Q”:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

constant integer base = part * pairs;

for p = 0 to pairs-1
    Elem[result, 2*p+0, esize] = Elem[operand1, base+p, esize];
    Elem[result, 2*p+1, esize] = Elem[operand2, base+p, esize];

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C7.2.386 ZIP2

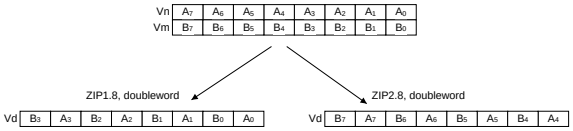
Zip vectors (secondary)

This instruction reads adjacent vector elements from the upper half of two source SIMD&FP registers as pairs, interleaves the pairs and places them into a vector, and writes the vector to the destination SIMD&FP register. The first pair from the first source register is placed into the two lowest vector elements, with subsequent pairs taken alternately from each source register.

Note

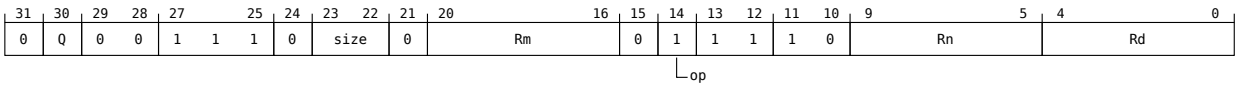
This instruction can be used with `ZIP1` to interleave two vectors.

The following figure shows an example of the operation of ZIP1 and ZIP2 with the arrangement specifier 8B.



Depending on the settings in the CPACR_EL1 and CPTR_EL2 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Advanced SIMD
(FEAT_AdvSIMD)



Encoding

```
ZIP2 <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AdvSIMD) then EndOfDecode(Decode_UNDEF);
if size:Q == '110' then EndOfDecode(Decode_UNDEF);

constant integer d = UInt(Rd);
constant integer n = UInt(Rn);
constant integer m = UInt(Rm);
constant integer esize = 8 << UInt(size);
constant integer datasize = 64 << UInt(Q);
constant integer elements = datasize DIV esize;
constant integer part = UInt(op);
constant integer pairs = elements DIV 2;
```

Assembler Symbols

<**Vd**> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<**T**> Is an arrangement specifier, encoded in “size:Q”:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<**Vn**> Is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<**Vm**> Is the name of the second SIMD&FP source register, encoded in the “Rm” field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
constant bits(datasize) operand1 = V[n, datasize];
constant bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

constant integer base = part * pairs;

for p = 0 to pairs-1
    Elem[result, 2*p+0, esize] = Elem[operand1, base+p, esize];
    Elem[result, 2*p+1, esize] = Elem[operand2, base+p, esize];

V[d, datasize] = result;
```

Operational information

This instruction is a data-independent-time instruction as described in About PSTATE.DIT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Part D
The AArch64 System Level Architecture

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter D1
The AArch64 System Level Programmers' Model

This chapter describes the AArch64 system level programmers' model. It contains the following sections:

- Exception levels.
- Registers for instruction processing and exception handling.
- Exceptions.
- Process state, PSTATE.
- Resets and power domains.
- Mechanisms for entering a low-power state.
- Self-hosted debug.
- Event monitors.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D1.1 Exception levels

R_{HTTFQ} The architecture defines three *Exception levels*: EL0, EL1 and EL2.

R_{VNPYH} EL2 is the highest Exception level and EL0 is the lowest. Therefore, EL2 is higher than EL1, and EL1 is higher than EL0.

R_{SWGNJ} A PE must implement EL0, EL1 and EL2.

I_{WRRKQ} EL2 provides support for the virtualization of EL0 and EL1.

R_{CCQWK} Unprivileged execution is any execution that occurs at EL0.

R_{NZZNS} The current Exception level changes only when any of the following occur:

- Taking an exception.
- Returning from an exception.
- Processor reset.
- Exiting from Debug state.
- If in Debug state, executing a DCPSx instruction.
- If in Debug state, executing a DRPS instruction.

R_{XRQKF} The *target Exception level* is the Exception level to which an exception is taken.

R_{FFJBB} Each exception type has a target Exception level that is either:

- Implicit in the type of the exception.
- Defined by configuration bits in the System registers.

R_{TKYYF} An exception cannot be taken to EL0.

R_{QNTPB} An exception cannot cause entry to a lower Exception level.

D1.1.1 Execution state

R_{PCFKP} The architecture defines one Execution state, AArch64 state using 64-bit processing.

R_{BDMRT} If an Exception level is using AArch64, then all higher Exception levels are using AArch64.

D1.1.2 Security states

R_{RVPDR} The architecture defines one *Security state*, Secure state.

R_{NRXJT} The architecture defines the following *Physical Address* (PA) spaces:

- Secure.
- Non-secure.

I_{NDJCC} Memory assigned to the Secure physical address space cannot be read or modified by an agent associated with Non-secure state.

I_{YFMYJ} For details on the System Agent and Non-secure Protected PA spaces, see Granular Data Isolation.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D1.1.3 Effect of not implementing an Exception level

R_{ZSYWZ} SCR_EL3.NS has a notional value of 0, meaning that the Security state is Secure.

R_{KFGRW} An exception cannot be taken to an unimplemented Exception level. This means that exceptions will target either EL1 or EL2.

D1.1.3.1 Typical Exception level usage model

I_{WCCCB} The architecture does not specify what software uses which Exception level. Such choices are outside the scope of the architecture. However, the following is a common usage model for the Exception levels:

Exception level	Usage
EL0	Applications.
EL1	OS kernel and associated functions that are typically described as privileged.
EL2	Hypervisor.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D1.2 Registers for instruction processing and exception handling

I_{BCVRH} In the Arm architecture, registers are split into two main categories:

- Registers that provide system control or status reporting.
- Registers that are used in instruction processing, for example to accumulate a result, and in handling exceptions.

For more information, see AArch64 System Register Descriptions.

D1.2.1 The general-purpose registers

R_{JBKTZ} The general-purpose register bank comprises the 31 general-purpose registers, R0-R30.

I_{RKTLT} The general-purpose register bank is associated with instructions in the base instruction set.

R_{YDKHJ} The general-purpose registers can be accessed as either of the following:

- 31 64-bit registers, X0-X30.
- 31 32-bit registers, W0-W30.

I_{PVGXY} When a Warm reset is asserted, the general-purpose registers are reset to an architecturally UNKNOWN value.
For more information, see Register size and Registers in AArch64 Execution state.

D1.2.2 The stack pointer registers

R_{VSNWX} Each Exception level has a dedicated stack pointer register. The stack pointer registers are:

- SP_EL0.
- SP_EL1.
- SP_EL2.

I_{YPMHQ} When a Warm reset is asserted, the stack pointers are reset to an architecturally UNKNOWN value.
For more information, see Special purpose registers and PC alignment checking.

D1.2.2.1 Stack pointer register selection

R_{LQMQQ} The PE uses the following stack pointers:

- If executing at EL0, then the PE uses the EL0 stack pointer, SP_EL0.
- If executing at EL1 or EL2, then the PE uses the stack pointer determined by PSTATE.SP:
  - If PSTATE.SP is 0, then the PE uses the EL0 stack pointer, SP_EL0.
  - If PSTATE.SP is 1, then the PE uses the stack pointer for the current Exception level.

I_{CSFMY} When an exception is taken, the stack pointer for the target Exception level is selected.

I_{VYNZY} The selected stack pointer can be indicated by a suffix to the Exception level:

- The t suffix, referring to thread, indicates use of the SP_EL0 stack pointer.
- The h suffix, referring to handler, indicates use of the SP_ELx stack pointer.

I_{QZDYD} The following are the AArch64 stack pointer options:

Exception level (EL)	Stack pointer (SP) option
EL0	SP_EL0
EL1	SP_EL1, SP_EL0

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Exception level (EL)	Stack pointer (SP) option
EL2	SP_EL2, SP_EL0

D1.2.3 The SIMD and floating-point registers

R_{JSCYH} The SIMD and floating-point register bank comprises the 32 quadword (128-bit) SIMD and floating-point registers, V0-V31.

I_{LMMMW} The SIMD and floating-point register bank is used for floating-point, vector, and other SIMD-related scalar operations.

R_{LFCJZ} The SIMD and floating-point registers can be accessed as any of the following:

- 32 quadword (128-bit) registers, Q0-Q31.
- 32 doubleword (64-bit) registers, D0-D31.
- 32 word (32-bit) registers, S0-S31.
- 32 halfword (16-bit) registers, H0-H31.
- 32 byte (8-bit) registers, B0-B31.
- 128-bit vectors of elements.
- 64-bit vectors of elements.

I_{KFYJJ} When a Warm reset is asserted, the SIMD and floating-point registers are reset to an architecturally UNKNOWN value.
For more information, see Registers in AArch64 Execution state.

D1.2.4 Saved Program Status Registers

R_{PJSSD} In AArch64 state, each implemented Exception level to which an exception can be taken has a Saved Program Status Register (SPSR). The following are the SPSRs:

- For exceptions taken to EL1 using AArch64, SPSR_EL1.
- For exceptions taken to EL2 using AArch64, SPSR_EL2.

I_{DVBPS} When a Warm reset is asserted, the SPSRs are reset to an architecturally UNKNOWN value.
For more information, see Process state, PSTATE.

D1.2.5 Exception Link Registers

R_{JQBYN} Exception Link Registers hold preferred exception return addresses.

R_{XFSNH} In AArch64 state, each implemented Exception level to which an exception can be taken has an Exception Link Register (ELR). The following are the ELRs:

- For exceptions taken to EL1, ELR_EL1.
- For exceptions taken to EL2, ELR_EL2.

I_{SHJSC} When a Warm reset is asserted, the ELRs are reset to an architecturally UNKNOWN value.
For more information, see Preferred exception return address.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D1.3 Exceptions

D1.3.1 Exception entry terminology

D1.3.1.1 Taken, taken from, and taken to

R_{VDFHD} When the PE responds to an exception, an exception is *taken*.

R_{FXGFX} The PE state immediately before taking the exception is the state the exception is *taken from*.

R_{WTMVN} The PE state immediately after taking the exception is the state the exception is *taken to*.

D1.3.1.2 Exception generating instructions

R_{QPPFN} *Exception generating instructions* intentionally produce a precise exception in the instruction stream immediately after the exception generating instruction. They are:

- HVC.
- SMC.
- SVC.

D1.3.1.3 Synchronous and asynchronous exceptions

R_{FQHGR} If all of the following apply, an exception is *synchronous*:

- The exception is generated as a result of direct execution or attempted execution of an instruction.
- The preferred exception return address has an architecturally defined relationship with the instruction that caused the exception.
- The exception is precise.

R_{PCXDH} An exception is *asynchronous* if it is not synchronous.

R_{JPZPR} Asynchronous exceptions taken to AArch64 state are also known as interrupts.

D1.3.1.4 Definition of a precise exception and imprecise exception

R_{HVZKH} An exception is *precise* if on taking the exception, the PE state and the memory system state is consistent with the PE having executed all of the instructions up to but not including the point in the instruction stream where the exception was taken from, and none afterwards. However, for an instruction executing immediately at the point in the instruction stream that the exception was taken from, the definition of precise also permits any of the following:

- For synchronous Data Abort and Watchpoint exceptions that are taken to AArch64 state generated by an instruction that performs more than one single-copy atomic memory access, the values in registers or memory affected by the instructions can be UNKNOWN, if all of the following apply:
  - The accesses affecting those registers or memory locations do not, themselves, generate exceptions or debug events.
  - The registers are not involved in the calculation of the memory address used by the instruction.
- For synchronous Data Abort and Watchpoint exceptions that are generated from load or store instructions executed in AArch64 state, all the following can occur:
  - If the instruction was a load to either the base address register or the offset register, that register is restored to the original value, and any other destination registers become UNKNOWN.
  - If the instruction was a load that does not load the base address register or the offset register, then the destination registers become UNKNOWN.
  - If the instruction was an atomic that attempted both a load and a store, and the load was to the base address register, a source register or a compare register, that register is restored to the original value. Otherwise, that destination register becomes UNKNOWN.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- For implementations that include synchronous exception generation for floating-point exceptions, when a floating-point exception is taken, it is permitted that the cumulative floating-point exception bits are not restored.
- For a precise exception that is taken from AArch64 state during an instruction that performs more than one single-copy atomic memory access, the values in registers or memory affected by the instructions can be UNKNOWN, if all of the following apply:
  - The accesses affecting those registers or memory locations do not, themselves, generate exceptions or debug events.
  - The registers are not involved in the calculation of the memory address used by the instruction.

R_{ZNCXP} An exception is *imprecise* if it is not a precise exception.

R_{LSGJD} Except for SError exceptions, all exceptions taken to AArch64 state are precise. For each occurrence of an SError exception, whether the interrupt is precise or imprecise is IMPLEMENTATION DEFINED.

I_{ZNNNY} An asynchronous Data Abort exception generated by a load that causes an SError exception to be taken at some point later in the instruction stream than the load that generated the exception, is usually imprecise. The SError exception is usually imprecise because the data returned from the load is UNKNOWN, and so can have corrupted the state that is presented at the time that the exception is taken.

D1.3.1.5 Preferred exception return address

I_{LQPWH} The *preferred exception return address* is an address that software might return to after handling an exception in order to resume execution.

I_{HBQRX} The preferred exception return address is determined by the type of the exception.

R_{DPLYN} For an exception taken to an Exception level, ELx, using AArch64, the Exception Link Register for that Exception level, ELR_ELx, is set to the preferred exception return address.

R_{VBQMV} For asynchronous exceptions, the preferred exception return address is the address of the instruction following the instruction boundary at which the interrupt occurs.

R_{QYCWH} For synchronous exceptions other than exception generating instructions, the preferred exception return address is the address of the instruction that generates the exception.

R_{DKWPP} For an exception generating instruction that is executed, the preferred exception return address is the address of the instruction that follows the exception generating instruction.

R_{LBLBR} For an exception generating instruction that is trapped, disabled, or is UNDEFINED because the Exception level has insufficient privilege to execute the instruction, the preferred exception return address is the address of the exception generating instruction.

D1.3.1.6 Exception vectors

R_{FKKHH} When an exception is taken to an Exception level that is using AArch64, execution starts at the *exception vector*.

I_{QLHPV} The memory at the exception vector for an exception is expected to contain the handler code that corresponds to that exception category.

R_{BRCKV} The vector base address for an Exception level, ELx, is defined by the Vector Base Address Register, VBAR_ELx.

R_{PJKRC} Each exception category has an exception vector at a fixed offset from the vector base address.

I_{YCTHH} Exceptions taken to AArch64 are categorized for the purpose of assigning exception vectors based on the following information:

- Whether the exception is one of the following:
  - Synchronous exception, excluding synchronous External aborts.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - Synchronous External abort.
  - SError exceptions, including vSErrors and delegated SError exceptions.
  - IRQ or vIRQ interrupts.
  - FIQ or vFIQ interrupts.
- Information about the Exception level that the exception came from.
- Information about the SP in use.
- The state of the register file.

For information on synchronous exceptions, see Synchronous exception types. For information on asynchronous exceptions, see Asynchronous exception types.

R_{RYXCL} The following tables describe the offsets that are added to the vector base address to describe the exception vector:

Exception taken from	Offset for exception type Synchronous exceptions, excluding External aborts	IRQ and vIRQ	FIQ and vFIQ
Current Exception level with SP_EL0.	0x000	0x080	0x100
Current Exception level with SP_ELx, x > 0.	0x200	0x280	0x300
Lower Exception level, where the implemented level immediately lower than the target level is using AArch64.	0x400	0x480	0x500

Exception taken from	Offset for exception type Synchronous External aborts	SError and vSError
Current Exception level with SP_EL0.	0x000	0x180
Current Exception level with SP_ELx, x > 0.	0x200	0x380
Lower Exception level, where the implemented level immediately lower than the target level is using AArch64.	0x400	0x580

D1.3.2 Exception entry

R_{DQXFW} When an exception is taken to an Exception level, ELx, that is using AArch64 state, all the following occur:

- Fields in SPSR_ELx are written according to their description in SPSR_ELx, based on the values in PSTATE immediately before the exception was taken.
- The preferred exception return address is written to ELR_ELx.
- The contents of PSTATE immediately after the exception is taken is as described in rule R_{YVRLT}.
- For synchronous exceptions and SError exceptions, exception syndrome information is written to ESR_ELx.
- Execution starts from the exception vector at the target Exception level.

R_{YVRLT} When an exception is taken to an Exception level, ELx, that is using AArch64 state, after recording the previous values in SPSR_ELx, the following PSTATE bits are set:

- PSTATE.EL is set to the target Exception level.
- All of PSTATE.{D, A, I, F, SP} are set to 1.
- PSTATE.{IL, nRW} are set to 0.
- PSTATE.SS is set according to the rules in AArch64 Self-hosted Debug.
- If FEAT_PAN is implemented, then for any of the following situations, PSTATE.PAN is set to 1:
  - The target Exception level is EL1 and SCTLR_EL1.SPAN is 0.
  - The exception is taken from EL0, the target Exception level is EL2 using AArch64, the Effective value of HCR_EL2.{TGE, E2H} is {1, 1}, and SCTLR_EL2.SPAN is 0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If FEAT_UAO is implemented, then PSTATE.UAO is set to 0.
- If FEAT_SSBS is implemented, then PSTATE.SSBS is set to the value of SCTLR_ELx.DSSBS.

R_{SRXVW} If the Effective value of SCTLR_ELx.IESB is 1 at the target Exception level, an exception taken to AArch64 state is an error synchronization event.

R_{GKPZX} Exception entry is a Context synchronization event. For the purposes of the memory model, exception entry is an Instruction Fetch Barrier effect.

I_{FZTLH} The ordering of error synchronization and context synchronization on exception entry is described in the pseudocode.

I_{LBHFM} See also Taking error exceptions.

D1.3.2.1 Synchronous exception entry

I_{VWGYJ} For synchronous exceptions that result in the exception handler emulating the instruction and returning to the following instruction, the handler typically completes by ensuring the state of the PE on returning from the exception is consistent with the emulated instruction having been executed. This usually includes ensuring the value of SPSR_ELx.BTYPE is consistent with the instruction being emulated having executed.

For SVCand HVC instructions, when these instructions execute without being trapped, the value of SPSR_ELx already points to the following instruction. However SPSR_ELx.BTYPE is not guaranteed to be 0 and so might not be consistent with the SVC or HVC instruction having executed. This might subsequently cause a Branch Target exception at the instruction after the SVC or HVC instruction.

R_{ZTSSH} For any of the following synchronous exception types, when an exception is taken to an Exception level, ELx, a *virtual address* (VA) that characterizes the exception is captured in FAR_ELx:

- An Instruction Abort exception.
- A Data Abort exception.
- A PC alignment fault exception.
- A Watchpoint exception.

For synchronous External aborts that are not caused by translation table walks, it is CONSTRAINED UNPREDICTABLE whether the FAR_ELx contains a VA that characterizes the exception. The ESR_ELx.FnV bit in the ISS encoding for the External abort indicates the validity of the VA in FAR_ELx:

- If ESR_ELx.FnV in the ISS encoding for the External abort is 1, the VA in FAR_ELx is UNKNOWN.
- If ESR_ELx.FnV in the ISS encoding for the External abort is 0, the VA in FAR_ELx is valid.

For all other exceptions taken to ELx, FAR_ELx is UNKNOWN.

R_{YJHVH} For Instruction Abort or Data Abort exceptions caused by any of the following faults, when an exception is taken to EL2, an *intermediate physical address* (IPA) that characterizes the exception is captured in HPFAR_EL2:

- A Translation fault, Access Flag fault, or Address Size fault on a stage 2 translation not on a stage 1 translation table walk.
- A Translation fault, Access flag fault, Address Size fault, or Permission fault on stage 2 translation of an address accessed in a stage 1 translation table walk.

For all other exceptions taken to EL2 using AArch64, HPFAR_EL2 is UNKNOWN.

R_{TGQRC} For Instruction Abort or Data Abort exceptions caused by an External abort, when FEAT_PFAR is implemented:

- If all of the following apply, then ESR_ELx.PFV is set to 0:
  - The exception is taken to EL1.
  - EL2 is implemented and enabled in the current Security state.
  - The Effective value of HCR_EL2.VM is 1.
- Otherwise, ESR_ELx.PFV is set to an IMPLEMENTATION DEFINED value of 0 or 1.

For all other Instruction Abort and Data Abort exceptions, ESR_ELx.PFV is set to 0.

On taking an Instruction Abort or Data Abort exception:

- If ESR_ELx.PFV is set to 1 by the PE, then a physical address space that characterizes the exception and a PA are captured in PFAR_ELx. The fault granule size is defined by D_{LVGRB}.
- If ESR_ELx.PFV is set to 0 by the PE, then the applicable PFAR_ELx register is set to an UNKNOWN value.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

PFAR_ELx never records the Intermediate Physical Address (IPA). PFAR_ELx might reveal a faulting physical addresses to a guest operating system if stage 2 translation is not being used and some other method is used to hide physical addresses from the guest (such as shadow page tables).

R_{YZKWJ} For all exceptions other than Instruction Abort and Data Abort exceptions, the applicable PFAR_ELx register is set to an UNKNOWN value.

D1.3.2.2 Asynchronous exception entry

R_{JBKRW} If any of the following apply, when a physical SError exception is taken to AArch64 state, the pending state of the physical SError is cleared:

- FEAT_DoubleFault is implemented.
- If FEAT_RAS is implemented, and on taking the SError exception, the syndrome recorded in ESR_ELx indicates an SError other than IMPLEMENTATION DEFINED or uncategorized SError exception syndrome. See Taking error exceptions.

Otherwise, it is IMPLEMENTATION DEFINED whether the pending state of the physical SError is cleared. This IMPLEMENTATION DEFINED behavior might vary according to the type of the SError exception.

R_{SYZRH} When a virtual SError exception is taken to AArch64 state, HCR_EL2.VSE is set to 0.

D1.3.3 Exception return terminology

D1.3.3.1 Return, return from, return to

R_{MMSFW} An exception return is caused by the execution of an exception return instruction.

R_{JDJHH} The PE state immediately before an exception return instruction is executed is the state the exception *returns from*.

R_{HNVMN} The PE state immediately after the execution of an exception return instruction is the state the exception *returns to*.

D1.3.4 Exception return

R_{SKNJF} In AArch64 state, the Exception return instructions are ERET, ERETAA, and ERETAB.

R_{PKJFB} The Exception return instructions are UNDEFINED in EL0.

R_{SVBYH} An exception return is either legal or illegal.

D1.3.4.1 Legal exception returns from AArch64 state

R_{BWCFK} On a legal exception return from an Exception level, ELx, all of the following occur:

- If SCTLR_ELx.IESB is 1, and the Exception return instruction does not generate an exception, the PE inserts an error synchronization event before the Exception return instruction.
- Fields in PSTATE that have a corresponding field in SPSR_ELx are written according to their description in SPSR_ELx.
- The PC is set to the value in ELR_ELx.
- The Event Register for the PE executing the Exception return instruction is set.
- The local Exclusives monitor for the PE executing the Exception return instruction is cleared. It is IMPLEMENTATION DEFINED whether clearing the local Exclusives monitor also clears the global Exclusives monitor.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D1.3.4.2 Illegal exception returns from AArch64 state

R_{FHSGL} If in AArch64 state, any of the following situations can cause an illegal exception return:

- A return is made to an Exception level higher than the current Exception level.
- A return is made to EL1 HCR_EL2.TGE is 1.
- A return where the SPSR_ELx.M[4] bit is 1.
- A return where the SPSR_ELx.M[3:0] bits are set to a reserved value.

R_{HWWQH} On an illegal exception return from an Exception level, ELx, all of the following occur:

- If SCTLR_ELx.IESB is 1, the PE inserts an error synchronization event before the Exception return instruction.
- The PC is set to the value in ELR_ELx.
- PSTATE.IL is set to 1.
- PSTATE.{EL, nRW, SP} are unchanged.
- PSTATE.{N, Z, C, V, D, A, I, F} are set to the associated values in SPSR_ELx.
- PSTATE.SS is handled as if the return is a legal exception return.
- If FEAT_PAN is implemented, then PSTATE.PAN is set to the associated value in SPSR_ELx.
- All of the following are UNKNOWN:
  - If FEAT_UAO is implemented, then PSTATE.UAO.
  - If FEAT_DIT is implemented, then PSTATE.DIT.
  - If FEAT_SSBS is implemented, then PSTATE.SSBS.
- The Event Register for the PE executing the Exception return instruction is set.
- The local Exclusives monitor for the PE executing the Exception return instruction is cleared. It is IMPLEMENTATION DEFINED whether clearing the local Exclusives monitor also clears the global Exclusives monitor.
- After the PC has been set to the value held in ELR_ELx and the contents of PSTATE have been set to the value held in SPSR_ELx, the ELR_ELx and SPSR_ELx become UNKNOWN.

D1.3.5 Synchronous exception types

R_{FGCQD} All of the following are synchronous exceptions:

- Any exception generated by attempting to execute an instruction that is UNDEFINED, including:
  - Attempts to execute instructions at an inappropriate Exception level.
  - Attempts to execute instructions when they are disabled.
  - Attempts to execute instruction bit patterns that are not allocated.
- Any exception caused by attempts to execute an instruction when the value of PSTATE.IL is 1. These are called Illegal Execution State exceptions.
- Any exception caused by the use of a misaligned SP. These are called SP Alignment Fault exceptions.
- Any exception caused by attempting to execute an instruction with a misaligned PC. These are called PC Alignment Fault exceptions.
- Any exception caused by a pointer authentication instruction authentication failure. These are called PAC Fail exceptions.
- Any exception caused by the exception generating instructions SVC or HVC. These are respectively called Supervisor Call (SVC) exceptions or Hypervisor Call (HVC) exceptions.
- Traps on attempts to execute instructions that the System registers define as instructions that are trapped to a higher Exception level. These are called Trap exceptions.
- Any exception caused by Instruction Aborts that were generated by the memory address translation system that was associated with attempts to execute instructions from areas of memory that generate faults. These are called Instruction Abort exceptions.
- Any exception caused by Data Aborts that were generated by the memory address translation system that are associated with attempts to read or write memory that generate faults. These are called Data Abort exceptions.
- Any exception caused by Data Aborts because of a misaligned address. These are called Data Abort exceptions.
- All of the debug exceptions:
  - Breakpoint Instruction exceptions.
  - Breakpoint exceptions.
  - Watchpoint exceptions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - Software Step exceptions.
- In an implementation that supports the trapping of floating-point exceptions, any exception caused by trapped floating-point exceptions. These are called Trapped Floating-point exceptions.
- An IMPLEMENTATION DEFINED set of exceptions caused by External aborts.

I_{XRFSY} The architecture permits, but does not require, synchronous or asynchronous handling of External aborts.

I_{WRXPC} The RAS architecture specifies some situations that must be handled synchronously.

D1.3.5.1 Taking synchronous exceptions from EL0

R_{NBBHH} If EL2 is using AArch64 and the Effective value of HCR_EL2.TGE is 1, when any of the following exceptions are taken from EL0, they are taken to EL2, unless routed to a higher Exception level. If EL2 is using AArch64 and the Effective value of HCR_EL2.TGE is 0, when any of the following exceptions are taken from EL0, they are taken to EL1, unless routed to a higher Exception level:

- Stage 1 Data Abort.
- Stage 1 Instruction Abort.
- PC Alignment fault.
- SP Alignment fault.
- Trapped Floating-point exception.
- Supervisor Call.
- Undefined Instruction exception.
- PAC Fail exception.
- WFE trap.
- WFI trap.
- Advanced SIMD and floating-point Access trap.
- Synchronous External aborts.
- PMU exceptions.

If EL2 is using AArch64 and either HCR_EL2.TGE is 1 or MDCR_EL2.TDE is 1, when any of the following debug exceptions are taken from EL0, they are taken to EL2. If EL2 is using AArch64 and both HCR_EL2.TGE is 0 and MDCR_EL2.TDE is 0, when any of the following debug exceptions are taken from EL0, they are taken to EL1:

- Breakpoint exception.
- Software Breakpoint exception.
- Software Step exception.
- Watchpoint exception.

If EL2 is using AArch64 and either HCR_EL2.TGE is 1 or MDCR_EL2.TDE is 1, when a Vector Catch exception is taken from EL0, it is taken to EL2. If EL2 is using AArch64 and both HCR_EL2.TGE is 0 and MDCR_EL2.TDE is 0, a Vector Catch exception cannot be taken.

R_{JNBTN} For an exception that is taken from EL0 to EL2 because the value of HCR_EL2.TGE is 1:

- If the exception would have been reported in ESR_EL1 using any EC value other than `0x07`, then the EC value and corresponding ISS encoding that would have been used to report the exception in ESR_EL1 are used to report the exception in ESR_EL2.
- If the exception would have been reported in ESR_EL1 using the EC value `0x07`, then the EC value `0x00` and ISS encoding value `0x00` are reported in ESR_EL2.

D1.3.5.2 Exception levels for taking a synchronous External abort

R_{YYFRT} If all of the following are true, then a synchronous External abort exception is taken to EL2:

- The exception is taken from EL1 or EL0.
- Any of the following are true:
  - The Effective value of HCR_EL2.TEA is 1.
  - The exception was generated by an External abort.
  - The Effective value of HCR_EL2.TGE is 1. This can only occur at EL0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise, if FEAT_DoubleFault2 is implemented and all of the following are true, then a synchronous External abort exception is taken to EL2:

- EL2 is implemented, enabled in the current Security state, and using AArch64.
- The Effective value of HCRX_EL2.TMEA is 1.
- PSTATE.A is 1.
- The exception is taken from EL1.

Otherwise, all of the following apply:

- A synchronous External abort exception taken from EL2 is taken to EL2.
- When EL1 is using AArch64, synchronous External abort exception taken from EL1 or EL0 is taken to EL1 using AArch64.

I_{VLBJF} The terms used in the tables in I_{NZCZW} have the following meanings:

HCR If EL2 is using AArch64, this is the Effective value of a field in HCR_EL2.

PSTATE The Effective value of a field in PSTATE.

TGE The Effective value of HCR.TGE. If EL2 is not implemented, or if EL2 is disabled in the current Security state, then the Effective value of HCR.TGE is 0.

A The Effective value of PSTATE.A.

EL1 The exception is taken to EL1.

EL2 The exception is taken to EL2.

n/a Not applicable. The field does not exist in the register in this configuration or the Exception level is not accessible in this configuration.

I_{NZCZW} The following table summarizes the destination Exception level for a synchronous External abort exception taken from ELx, and all of the following applies:

- The error was generated by a stage 1 fault on an access that is not a u System register access. See R_{VLBJF} for details of other cases.
- Either EDSCR.INTdis is `0b00` or invasive debug is disabled.

HCR	PSTATE	Exception taken from		
TGE	A	EL0	EL1	EL2
`0b0`	`0b0`	EL1	EL1	EL2
`0b0`	`0b1`	EL1	EL2	EL2
`0b1`	`0b0`	EL2	n/a	EL2
`0b1`	`0b1`	EL2	n/a	EL2

D1.3.5.3 Prioritization of Synchronous exceptions taken to AArch64 state

R_{ZFGJP} The following list shows the priorities for synchronous exceptions taken to an Exception level using AArch64. The highest priority is 1.

Priority	Synchronous exception type
1	Reset Catch debug event. See Reset Catch debug events.
2	Exception Catch debug event if it has a priority of 2. See Exception Catch debug event.
3	Halting Step debug event. See Halting Step debug events.
4	Software Step Exception used during Debug. See Software Step exceptions.
5	Synchronous PMU exception. See Synchronous exception-based event profiling.
6	Exception Catch debug events. See Exception Catch debug event.
7	PC alignment fault exception. See PC alignment checking.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Priority	Synchronous exception type
8	N/A
9	Instruction Abort exceptions, including exceptions generated by an MMU or MPU fault for the translation of an instruction fetch. See - MMU fault prioritization from a single address translation stage. • MPU memory translations and faults
10	Breakpoint exceptions. See Breakpoint exceptions.
11	N/A
12	N/A
13	Software Breakpoint exceptions, caused by execution of a Breakpoint instruction.
14	N/A
15	N/A
16	Exceptions taken from EL1 to EL2 because of the configuration of one of the following: • HSTR_EL2.Tn. • HCR_EL2.TIDCP.
17	Exceptions that occur as a result of attempting to execute an instruction that is UNDEFINED for one or more of the following reasons: • Attempting to execute a `UDF` instruction. • Attempting to execute an unallocated instruction encoding, including an encoding for an instruction that is not implemented in the PE implementation. • Attempting to execute an instruction that is defined to never be accessible at the current Exception level and Security state, regardless of any enables or traps. • Debug state execution of an instruction encoding that is not accessible in Debug state. • Non-debug state execution of an instruction encoding that is not accessible in Non-debug state. • Attempting to execute an `HVC` instruction, when `HVC` instructions are disabled by HCR_EL2.HCD. • Attempting to execute an `MSR` or `MRS` instruction to SP_EL0, when SPSel.SP is 0. • Attempting to execute an `MSR` or `MRS` instruction using an _EL12 register name, when HCR_EL2.E2H is 0. • Attempting to execute an `HLT` instruction, when `HLT` instructions are disabled by EDSCR.HDE or halting is prohibited. • If in Debug state attempting to execute a `DCPS2` instruction in EL1 or EL0. • Attempting to execute an instruction that is UNDEFINED because all of the following apply: — Programming FPCR.{Stride, Len} to nonzero values is supported. — One or more of FPCR.{Stride, Len} is nonzero.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Priority	Synchronous exception type
18	Exceptions that occur as a result of attempting to execute an instruction that is UNDEFINED for one or more of the following reasons: • Attempting to execute a UDF instruction. • Attempting to execute an unallocated instruction encoding, including an encoding for an instruction that is not implemented in the PE implementation. • Attempting to execute an instruction that is defined to never be accessible at the current Exception level and Security state, regardless of any enables or traps. • Debug state execution of an instruction encoding that is not accessible in Debug state. • Non-debug state execution of an instruction encoding that is not accessible in Non-debug state. • Attempting to execute an HVC instruction, when HVC instructions are disabled by HCR_EL2.HCD. • Attempting to execute an MSR or MRS instruction to SP_EL0, when SPSel.SP is 0. • Attempting to execute an MSR or MRS instruction using an _EL12 register name, when the Effective value of HCR_EL2.E2H is 0. • If FEAT_FGT and FEAT_PMUv3 are implemented, attempting to execute an MSR or MRS instruction in AArch64 state,that accesses a register associated with an unimplemented PMU event counter. • Attempting to execute an HLT instruction, when HLT instructions are disabled by EDSCR.HDE or halting is prohibited. • If in Debug state: — Attempting to execute a DCPS1 instruction in Non-secure EL0 HCR_EL2.TGE is 1. — Attempting to execute a DCPS2 instruction in EL1 or EL0, when EL2 is disabled in the current Security state or is not implemented. • Attempting to execute an instruction that is UNDEFINED because all of the following apply: — EL1 is using AArch64. — Programming FPCR.{Stride, Len} to nonzero values is supported. — One or more of FPCR.{Stride, Len} is nonzero.
19	If all of the following apply, any exception that is generated by a configurable trap or enable: • The exception is not already covered by priorities 4-18. • The exception is taken to EL1, or, if HCR_EL2.TGE is 1, is taken to EL2.
20	As the result of one of the following configuration controls, an exception taken from EL0 to EL2: • When the exception is taken to AArch64 state: — HSTR_EL2.T*n*. — HCR_EL2.TIDCP.
21	N/A
22	As the result of one of the following configuration controls, an exception taken to EL2: • When the exception is taken to AArch64 state: CPTR_EL2, other than those controls covered in priority 21.
23	As the result of a configuration control in one of the following registers, an exception taken to EL2: • When the exception is taken to AArch64 state: — HCR_EL2: Other than a setting made in the {TIDCP, NV} fields, and MRS/MSR instruction using an _EL12 register name when HCR_EL2.E2H is 0. — CNTHCTL_EL2: Any setting. — MDCR_EL2: Any setting.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Priority	Synchronous exception type
24	Other than an exception defined by priorities 4-23 inclusive, any exception that is the result of a configurable access to instructions, where the exception is taken to EL2.
25	N/A
26	An exception caused by any of the following Exception generating instructions: • HVC. • SVC. For AArch64 State, see Branches, Exception generating, and System instructions.
27	N/A
28	N/A
29	N/A
30	N/A
31	N/A
32	N/A
33	N/A
34	N/A
35	N/A
36	When FEAT_FPAC is implemented, a PAC Fail exception. See Faulting on pointer authentication.
37	When SIMD & Floating Point is implemented, any trapped floating-point exception. See Floating-point exceptions and exception traps.
38	This priority number is used by debug events.
39	An SP alignment fault. See SP alignment checking.
40	N/A
41	N/A
42	N/A
43	N/A
44	Watchpoint exception. See Watchpoint exceptions.
45	N/A
46	N/A

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D1.3.5.4 Trapping of floating-point exceptions

R_{TSQFJ} Execution of a floating-point instruction, or execution of an Advanced SIMD instruction that performs floating-point operations, can generate an exceptional condition, called a floating-point exception.

R_{BBSGN} For each of the following floating-point exceptions, it is IMPLEMENTATION DEFINED whether an implementation includes synchronous exception generation:

- Input Denormal.
- Inexact.
- Underflow.
- Overflow.
- Divide by Zero.
- Invalid Operation.

I_{YRVSW} The architecture does not support asynchronous reporting of floating-point exceptions. If an implementation does not support synchronous exception generation from a floating-point exception, then that synchronous exception is never generated and all statements on when that synchronous exception is generated do not apply.

I_{KJTLV} For any of the implemented floating-point exceptions listed in R_{BBSGN}, FPCR provides control bits to enable synchronous exception generation.

R_{GXMQK} The Exception level that a Trapped Floating-point exception is taken to is defined as follows:

- If executing at EL0:
  - If the Effective value of HCR_EL2.TGE is 0, the exception is taken to EL1.
  - If the Effective value of HCR_EL2.TGE is 1, the exception is taken to EL2.
- If executing at EL1, the exception is taken to EL1.
- If executing at EL2, the exception is taken to EL2.

R_{TRWMV} If an implementation includes synchronous exception generation for floating-point exceptions in AArch64 state, when the execution of separate operations in separate SIMD elements causes multiple floating-point exceptions, the ESR_ELx reports only the exceptions associated with one element that the instruction uses. The architecture does not specify which element is reported.

R_{VJNYH} When a floating-point exception is trapped, all of the following apply:

- When the trapped floating-point exception is taken, it is IMPLEMENTATION DEFINED whether the FPSR is restored to the value of the FPSR immediately before the instruction that generated the trapped floating-point exception.When the trapped floating-point exception is taken, if the FPSR is not restored, it is CONSTRAINED UNPREDICTABLE which untrapped floating-point exceptions, if any, are indicated by the corresponding FPSR cumulative floating-point exception bits having the value 1.
- In ESR_ELx at the target Exception level, all of the following apply:
  - The highest priority trapped floating-point exception has a floating-point exception trapped bit set to 1.
  - If any other untrapped floating-point exceptions are generated by the same operation, each untrapped exception has a floating-point exception trapped bit set to 0. This applies to both higher priority and lower priority untrapped floating-point exceptions.
  - If any lower priority trapped floating-point exceptions are generated by the same operation, for each exception, it is CONSTRAINED UNPREDICTABLE whether the floating-point exception trapped bit is set to 1.

R_{FTPRW} For trapped floating-point exceptions from Advanced SIMD instructions, the architecture does not define the floating-point exception prioritization between different elements of the instruction.

D1.3.6 Asynchronous exception types

R_{ZNTKG} The Arm architecture Exception model distinguishes the following classes of asynchronous exceptions:

- Physical Interrupts
- Virtual Interrupts

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- PMU exceptions, see Exception-based event profiling.

There are the following types of physical interrupt:

- SError (also described as a System Error)
- IRQ
- FIQ

There are the following types of virtual interrupt:

- vSError (also described as a Virtual System Error)
- Delegated SError
- vIRQ
- vFIQ

R_{SLLMN} IRQ, FIQ, vIRQ, and vFIQ interrupts are precise asynchronous exceptions.

R_{PBKNX} The mechanism by which Superpriority is assigned is IMPLEMENTATION DEFINED.

R_{DXHJZ} Each physical interrupt type can be assigned a target Exception level of EL1 or EL2.

I_{VMWYT} ISR_EL1 shows the pending status of the physical interrupts, the physical SError exception, the virtual interrupts, and the virtual and delegated SError exceptions, as follows:

- When read from EL1:
  - The pending status of whichever virtual interrupts are enabled, and of the virtual SError exception if enabled. If the virtual SError exception is not enabled and the delegated SError exception is enabled, the pending status of the delegated SError exception.
- When read from EL2:
  - The pending status of the delegated SError exception, if the delegated SError exception is enabled.

Otherwise, when read from EL1, EL2, or EL3, the pending status of the physical interrupts, and of the physical SError exception.

I_{RWYFW} An implementation might support other mechanisms for signaling a virtual interrupt.

I_{WRGNC} PMU exceptions can also be an asynchronous exception type. See Exception-based event profiling.

I_{XHCGZ} The mechanism by which physical interrupts are signaled to the PE are IMPLEMENTATION DEFINED and might be either edge or level sensitive. A common implementation choice is that the IRQ and FIQ interrupts are level sensitive, and this is supported by the Generic Interrupt Controller (GIC).

I_{YSKZH} The physical SError exception is often used, amongst other things, for communicating External aborts from the memory system that are to be taken asynchronously.

R_{KTWMT} For an External abort generated by the memory system that is taken asynchronously using the SError exception, the SError exception always behaves as an edge-triggered interrupt. For any other sources of SError exceptions, it is IMPLEMENTATION DEFINED whether they are edge-triggered or level-sensitive.

R_{VVBQF} When taking the SError exception to an Exception level using AArch64, ESR_ELx at the target Exception level is updated to describe the SError exception.

If FEAT_RAS is implemented, then when taking a vSError exception to an Exception level using AArch64, ESR_ELx at the target Exception level is updated with exception syndrome information from VSESR_EL2.

R_{RKDNV} When taking an IRQ, vIRQ, FIQ or vFIQ interrupt to an Exception level using AArch64, ESR_ELx at the target Exception level is not updated.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D1.3.6.1 Virtual interrupts

R_{MLLVN} Each virtual interrupt type can be independently enabled from EL2. If a virtual interrupt type is enabled from EL2, the target Exception level for the corresponding physical interrupt is not EL1.

R_{BCXJB} If HCR_EL2.TGE is 0, then setting an HCR_EL2.{FMO, IMO} routing control bit to 1 enables the corresponding virtual interrupt.
If HCR_EL2.TGE is 1, virtual interrupts are disabled.

R_{CDCKC} If HCR_EL2.TGE is 0, then setting the Effective value of either the HCR_EL2.AMO or HCRX_EL2.TMEA routing control bit to 1 enables the virtual SError exception.
If HCR_EL2.TGE is 1, then virtual SError exceptions are disabled.

R_{NVFXB} If a virtual interrupt type is enabled, that type of interrupt can be generated by any one of the following:

- Execution at EL0 or EL1 if the corresponding virtual interrupt pending bit, HCR_EL2.{VSE, VI, VF}, is 1.
- For a vIRQ or a vFIQ, by an IMPLEMENTATION DEFINED mechanism. This might be a signal from an interrupt controller.

R_{SDVLM} If a virtual interrupt is disabled, the virtual interrupt cannot be taken.

R_{MZWNH} When taking a vIRQ or a vFIQ interrupt, the corresponding virtual interrupt pending bit in HCR_EL2 retains its state.

I_{QVCWH} If the virtual interrupt pending bits are used, the vIRQ or vFIQ exception handler must cause software executing at EL2 to set their corresponding virtual interrupt pending bits to 0.

I_{YPRNB} When taking a vSError exception, HCR_EL2.VSE is cleared to 0.

D1.3.6.2 Establishing the target Exception level of an asynchronous exception

R_{QZPXL} The R_{NMMXK} table uses the following terms:

Term	Meaning
IRQ FIQ	The Effective value of these fields is 0.
TGE	If EL2 is not implemented, the Effective value of HCR_EL2.TGE is 0.
IMO FMO	If EL2 is not implemented, the Effective value of these fields is 0.
RW	HCR_EL2.RW
EL1	The exception is taken to EL1.
EL2	The exception is taken to EL2.
C	The interrupt is not taken and remains pending, regardless of the PSTATE.{I, F, ALLINT} interrupt masks.
n/a	Not applicable. The field does not exist in the register in this configuration or the Exception level is not accessible in this configuration.

R_{NMMXK} The following table describes the routing of physical FIQ and IRQ interrupts if the highest implemented Exception level is using AArch64 and either EDSCR.INTdis is `0b00` or invasive debug is disabled.

Table D1-8 FIQ and IRQ interrupt exception targets

SCR_EL3				Target when taken from		
NS	EEL2	IRQ FIQ	RW	EL0	EL1	EL2
0	1	0	x	EL1	EL1	C
0	1	0	x	EL1	EL1	C

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SCR_EL3				Target when taken from		
NS	EEL2	IRQ FIQ	RW	EL0	EL1	EL2
0	1	0	x	EL2	EL2	EL2
0	1	0	x	EL2	n/a	EL2
0	1	1	x	EL2	EL2	C
0	1	1	x	C	n/a	C

Table D1-9 FIQ and IRQ interrupt exception targets

HCR			Target when taken from		
TGE	IMO FMO	RW	EL0	EL1	EL2
0	0	0	IRQ, FIQ	IRQ, FIQ	C
0	0	1	EL1	EL1	C
0	0	x	EL1	EL1	C
0	1	x	EL2	EL2	EL2
1	x	x	EL2	n/a	EL2
0	0	0	FIQ	FIQ	C
0	0	1	EL1	EL1	C
0	0	x	EL1	EL1	C
0	1	x	EL2	EL2	EL2
1	x	x	EL2	n/a	EL2

R_{ZZYHK} The R_{MGNFZ} tables use the following terms:

Term	Meaning
HCR_EL2	The Effective value of a field in HCR_EL2.
TGE	If EL2 is not implemented, or if EL2 is disabled in the current Security state, then the Effective value of HCR_EL2.TGE is 0.
AMO	If EL2 is not implemented, or if EL2 is disabled in the current Security state, the Effective value of HCR_EL2.AMO is 0.
PSTATE	The value of a field in PSTATE.
A	The value of PSTATE.A.
EL1	The exception is taken to EL1.
EL2	The exception is taken to EL2.
C	The interrupt is not taken and remains pending, regardless of the PSTATE.A mask.
n/a	Not applicable. The field does not exist in the register in this configuration or the Exception level is not accessible in this configuration.

R_{MGNFZ} The following tables describe the routing of SError exceptions taken from ELx.

All of the following apply to these tables:

- SCR_EL3.{NS, EEL2} are not shown because R_{ZZYHK} describes Effective values involving Security states.
- Either EDSCR.INTdis is `0b00` or invasive debug is disabled.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D1-11 SError exception target when EL2 is using AArch64

HCR_EL2		PSTATE	Taken from		
TGE	AMO	A	EL0	EL1	EL2
0b0	0b0	x	EL1	EL1	C
0b0	0b0	0b1	EL2	EL2	C
0b0	0b0	0b0	EL1	EL1	C
0b0	0b0	0b1	EL1	EL2	C
0b0	0b0	0b1	EL2	EL2	EL2
0b1	x	x	EL2	n/a	EL2
0b0	0b0	0b1	EL1	EL2	EL2
0b0	0b0	0b0	EL1	EL1	EL2
0b0	0b1	0b0	EL2	EL2	EL2
0b0	0b1	0b1	EL2	EL2	EL2
0b1	x	0b1	EL2	n/a	EL2
0b1	x	0b0	EL2	n/a	EL2

D1.3.6.3 Asynchronous exception masking

R_{HDMYQ} An interrupt can be masked. A masked interrupt is not taken and remains pending.

R_{ZZBPC} Whether an interrupt is masked is determined by all of the following:

- The PSTATE.{A, I, F, SP} bits.
- Whether an IRQ or FIQ interrupt has Superpriority.
- The implemented Exception levels.
- The configuration of the Hypervisor Configuration Registers.
- For an SError, whether the exception is non-maskable.
- The configuration of the Secure Configuration Register.

R_{LMWZH} If in AArch64 state and the target Exception level of an interrupt is lower than the current Exception level, the interrupt is masked.

D1.3.6.3.1 Physical interrupt masking

R_{MHWBP} If the target Exception level of a physical IRQ interrupt is the current Exception level, ELx, the following controls determine whether the interrupt is masked:

PSTATE.I	IRQ
0	Not masked
1	Masked

R_{GFXKY} If the target Exception level of a physical FIQ interrupt is the current Exception level, ELx, the following controls determine whether the interrupt is masked:

R_{YSVFS} If the target Exception level of a physical SError is the current Exception level, the PSTATE.A control determines whether the interrupt is masked.

R_{XCFNZ} If the target Exception level of a physical interrupt is higher than the current Exception level, all of the following apply:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the target Exception level is EL2 and any of the following are true, the interrupt cannot be masked by the PSTATE.{A, I, F} bits.
  - The Effective value of HCR_EL2.TGE is 0.
- If the target Exception level is EL2 and all of the following are true, the interrupt can be masked by the PSTATE.{A, I, F} bits:
  - The Effective value of HCR_EL2.TGE is 1.

I_{KDYCN} The ability to execute in EL0 with interrupts taken to EL1 masked is required by some user level driver code.

R_{HPRNX} The R_{SXLWJ} table uses the following terms:

Term	Meaning
A	When the interrupt is pending, it is taken regardless of the value of the PSTATE.{A, I, F, ALLINT} interrupt masks.
B	When the interrupt is pending, it might be subject to masking, as defined in R_{MHWBP} or R_{GFXKY}. If the interrupt is masked, it is not taken. If the interrupt is not masked, it is taken.
C	When the interrupt is pending, it is not taken, regardless of the value of the PSTATE.{A, I, F} interrupt masks.
n/a	Not applicable.

R_{SXLWJ} The following table describes the target and masking of physical FIQ and IRQ interrupts, if the highest implemented Exception level is using AArch64.

Table D1-14 FIQ and IRQ interrupt exception targets and masking

HCR		Effect of the interrupt mask when executing at:		
TGE	IMO FMO	EL0	EL1	EL2
0	0	B	B	C
0	1	A	A	B
1	x	A	n/a	B
1	x	A	n/a	A

D1.3.6.3.2 Virtual interrupt masking

R_{SNLJH} The following controls determine whether a virtual IRQ interrupt is masked:

PSTATE.I	vIRQ
0	Not Masked
1	Masked

R_{XSPSG} The following controls determine whether a virtual FIQ interrupt is masked:

PSTATE.F	vFIQ
0	Not Masked
1	Masked

R_{GYGBD} Virtual interrupts can only be taken from EL0 or EL1 and can only be taken to EL1. If executing at EL2, all types of virtual interrupts are always masked.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D1.3.6.4 Prioritization of interrupts

R_{NLHMN} For all of the following Context synchronization events, if an interrupt is pending before the Context synchronization event, and the interrupt remains pending and is not masked after the Context synchronization event, the interrupt is taken before the first instruction after the Context synchronization event:

- Execution of an ISB instruction.
- Exception entry.
- Exception return.
- Exit from Debug state.

If the first instruction after the Context synchronization event generates a synchronous exception, the architecture does not define whether the PE takes the interrupt or the synchronous exception first.

I_{RDBWJ} A RAS error synchronization event defines additional requirements for taking an SError exception.

R_{QVRHH} Other than the behaviors described in R_{NLHMN}, an unmasked, pending interrupt must be taken in finite time.

R_{ZQPLL} If an unmasked interrupt was pending but is changed to not pending before it is taken, it is CONSTRAINED UNPREDICTABLE whether or not the interrupt is taken. If the interrupt is taken, it is taken before the first Context synchronization event after the interrupt was changed to be not pending.

D1.3.6.5 Taking an interrupt during a multi-access load or store

R_{ZBFSL} If in AArch64 state, interrupts can be taken during a sequence of memory accesses caused by a single load or store instruction. This is true regardless of the memory type being accessed.
In this situation, the behavior is consistent with the requirements described in R_{TNVSL} in Definition of a precise exception and imprecise exception.

D1.3.7 UNDEFINED instructions

R_{ZLJDF} An instruction which is UNDEFINED generates a synchronous exception for that instruction unless there is a higher priority exception generated for that instruction.

R_{QNGXB} The Exception level that the synchronous Undefined Instruction exception is taken to is defined as follows:

- If executing at EL0:
  - If the Effective value of HCR_EL2.TGE is 0, the exception is taken to EL1.
  - If the Effective value of HCR_EL2.TGE is 1, the exception is taken to EL2.
- If executing at EL1, the exception is taken to EL1.
- If executing at EL2, the exception is taken to EL2.

D1.3.8 Configurable instruction controls

R_{QMWPZ} *Configurable instruction controls* are control bits held in System registers that determine whether attempting to execute an instruction generates a synchronous exception at the point in the instruction stream of that instruction, and the instruction is not executed.

I_{ZSSHJ} Configurable instruction controls might be referred to by any of the following names:

- Instruction enables.
- Instruction disables.
- Trap controls.

The definitions of each type overlap, and in some cases is historical. Describing a register control field as an instruction enable, an instruction disable, or a trap control, gives no indication of how an exception that is generated as a consequence of the value of that field is handled or reported. Each configurable instruction control defines how the exception that is generated as a consequence of the configurable instruction control is handled or reported.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{VWWGJ} An exception can only be generated as a result of a configurable instruction control if all the following apply:

- The instruction generating the exception does not also generate a higher priority exception.
- The instruction generating the exception is not UNPREDICTABLE or CONSTRAINED UNPREDICTABLE in the PE state in which the instruction is executed.

R_{JTXTF} It is UNPREDICTABLE / CONSTRAINED UNPREDICTABLE whether configurable instruction controls generate an exception when the instruction is UNPREDICTABLE or CONSTRAINED UNPREDICTABLE in the PE state in which the instruction is executed, with all of the following constraints:

- If the instruction description explicitly states that the configurable instruction control is applied with higher priority than the CONSTRAINED UNPREDICTABLE behavior, then the configurable instruction control generates an exception.
- The CONSTRAINED UNPREDICTABLE behaviors cannot lead to any behavior that is prohibited by the general definition of UNPREDICTABLE.

I_{BJBMF} UNPREDICTABLE and CONSTRAINED UNPREDICTABLE instructions can generate exceptions as a result of configurable instruction controls, but the architecture does not require them to do so.

Note: Many CONSTRAINED UNPREDICTABLE behaviors for instructions include an allowance that the CONSTRAINED UNPREDICTABLE instruction behaves the same way as a closely related instruction that is not CONSTRAINED UNPREDICTABLE. In those cases, the instruction enable, disable, or trap control that causes in exception on the closely related instruction will cause the same exception on the CONSTRAINED UNPREDICTABLE instruction.

I_{TFGPW} An implementation might provide more controls, in IMPLEMENTATION DEFINED registers, to provide control of trapping of IMPLEMENTATION DEFINED features.

R_{QQLHG} When a configurable instruction control causes an exception, the exception is taken and the instruction is not executed, and therefore all the following are true:

- The preferred exception return address of the exception is the instruction that generates the exception.
- There are no changes to the registers accessed by the instruction, including as a result of side-effects of a register access.

D1.3.8.1 EL0 and EL1 configurable instruction controls

I_{TJPQP} The following EL0 and EL1 System registers contain configurable instruction controls:

Register name	Register description
CPACR_EL1	Architectural Feature Access Control Register
MDSCR_EL1	Monitor System Debug Control Register
PMUSERENR_EL0	Performance Monitors User Enable Register
SCTLR_EL1	System Control Register (EL1)
TCR_EL1	Translation Control Register (EL1)

D1.3.8.2 EL2 configurable instruction controls

I_{BLDYL} The following EL2 System registers contain configurable instruction controls:

Register name	Register description
CPTR_EL2	Architectural Feature Trap Register, EL2
HCR_EL2	Hypervisor Configuration Register
HSTR_EL2	Hypervisor System Trap Register

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register name	Register description
MDCR_EL2	Monitor Debug Configuration Register, EL2
SCTLR_EL2	System Control Register, EL2
TCR_EL2	Translation Control Register, EL2

D1.3.9 Exception generating instructions

I_{YRYNQ} The exception generating instructions are commonly called system calls and refer to the instructions that generate any of the following synchronous exception types:

- Supervisor Call exception.
- Hypervisor Call exception.
- Secure Monitor Call exception.

R_{JZKSX} If the PE is executing at EL0, the `HVC` instruction is UNDEFINED.

R_{BRCQB} A Supervisor Call is generated by executing an `SVC` instruction.

R_{MTFVD} A Supervisor Call exception is taken to the following Exception levels:

- If executing at EL0:
  - If the Effective value of HCR_EL2.TGE is 0, the exception is taken to EL1.
  - If the Effective value of HCR_EL2.TGE is 1, the exception is taken to EL2.
- If executing at EL1, the exception is taken to EL1.
- If executing at EL2, the exception is taken to EL2.

I_{NCTDB} A Supervisor Call enables software executing at EL0 to make a call to an operating system or other software executing at EL1.

R_{LLLGJ} If EL2 is implemented and the `HVC` instruction is not UNDEFINED, a Hypervisor Call is generated by executing an `HVC` instruction.

R_{LQBSH} A Hypervisor Call exception is taken to the following Exception levels:

- When taken from EL1, the exception is taken to EL2.
- When taken from EL2, the exception is taken to EL2.

R_{HMXQS} If any of the following are true, the `SMC` instruction is UNDEFINED:

- The PE is executing at EL0.
- The `SMC` instruction is not trapped to EL2 by HCR_EL2.TSC.

D1.3.10 Program Counter and stack pointer alignment

D1.3.10.1 PC alignment checking

R_{QCCWG} If bits [1:0] of the PC are not `0b00`, there is a misaligned PC.

R_{SJLDM} The execution of an instruction with a misaligned PC generates a synchronous PC Alignment exception on that instruction.

R_{THMSY} A PC Alignment exception is taken to the following Exception levels:

- If executing at EL0:
  - If HCR_EL2.TGE is 0, the exception is taken to EL1.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - If HCR_EL2.TGE is 1, the exception it taken to EL2.
- If executing at EL1, the exception is taken to EL1.
- If executing at EL2, the exception is taken to EL2.

R_{MVTQT} When a PC Alignment Fault exception is taken to an Exception level, ELx, using AArch64, the ELR_ELx and the FAR_ELx both hold the entire PC in its misaligned form.

I_{VFXPV} A misalignment of the PC is an indication of a serious error, for example software corruption of an address.

D1.3.10.2 SP alignment checking

R_{RDMXG} When the SP is used as the base address of a calculation, regardless of any offset applied by the instruction, if bits [3:0] of the SP are not `0b0000`, there is a misaligned SP.

R_{TFVSM} If SP alignment checking is enabled, then the execution of a load or store using the SP with a misaligned SP generates a synchronous SP Alignment exception on that load or store.

R_{JQXVP} PRFM instructions that use the SP do not perform stack alignment checking.

R_{WXBYR} The following bits enable SP alignment checking at each Exception level when that Exception level is using AArch64.

- SCTLR_EL1.SA controls EL1.
- SCTLR_EL2.SA controls EL2.

R_{GVQYS} An SP Alignment exception is taken to the following Exception levels:

- If executing at EL0:
  - If HCR_EL2.TGE is 0, the exception is taken to EL1
  - If HCR_EL2.TGE is 1, the exception it taken to EL2.
- If executing at EL1, the exception is taken to EL1.
- If executing at EL2, the exception is taken to EL2.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D1.4 Process state, PSTATE

R_{SPCZF} Process state, or PSTATE, is an abstraction of process state information.

I_{LPKMT} All Arm instruction sets provide instructions that operate on elements of PSTATE.

I_{ZHGST} PSTATE is defined in pseudocode as the PSTATE structure, of type `ProcState`.

D1.4.1 PSTATE fields that are meaningful in AArch64 state

I_{LLLXL} PSTATE fields that are meaningful in AArch64 state are grouped into the following categories:

- Condition flags.
- Execution state controls.
- Exception mask bits.
- Access control bits.
- Timing control bits.
- Speculation control bits.

R_{PCDTX} The following PSTATE bits are meaningful in AArch64 state:

PSTATE field name	PSTATE field group	Required implemented feature	Value when a Warm reset is asserted	Additional details
N, Negative condition flag	Condition flags	-	-	-
Z, Zero condition flag	Condition flags	-	-	-
C, Carry condition flag	Condition flags	-	-	-
V, Overflow condition flag	Condition flags	-	-	-
SS, Software Step bit	Execution state controls	-	0	-
IL, Illegal Execution state bit	Execution state controls	-	0	-
nRW, Current Execution state bit	Execution state controls	-	0	If the current Execution state is AArch64, the PSTATE.nRW bit is 0.
EL, Current Exception level field	Execution state controls	-	When a Warm reset is asserted into an Exception level using AArch64, the PSTATE.EL field holds the encoding for the highest implemented Exception level, EL2.	-
SP, Stack pointer register selection bit	Execution state controls	-	1	-
D, Debug exception mask bit	Exception mask bits	-	1	-
A, I, F Asynchronous exception mask bits	Exception mask bits	-	1	-
PAN, Privileged Access Never state bit	Access control bits	FEAT_PAN	-	-
UAO, User Access Override bit	Access control bits	FEAT_UAO	-	-
DIT, Data Independent Timing bit	Timing control bits	FEAT_DIT	0	-

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PSTATE field name	PSTATE field group	Required implemented feature	Value when a Warm reset is asserted	Additional details
SSBS, Speculative Store Bypass Safe bit	Speculation control bits	FEAT_SSBS	IMPLEMENTATION DEFINED	-

D1.4.1.1 Accessing PSTATE fields

R_{LYGBS} In AArch64 state, PSTATE fields are accessed using Special-purpose registers that are directly read using the MRS (register) instructions, and directly written using the MSR (register) instructions.

R_{DGPCL} If in AArch64 state, the following Special-purpose registers can access the PSTATE fields that hold AArch64 state:

Special-purpose register	PSTATE fields
NZCV	N, Z, C, V
DAIF	D, A, I, F
CurrentEL	EL
SPSel	SP
PAN	PAN
UAO	UAO
DIT	DIT
SSBS	SSBS

All other PSTATE fields do not have direct read and write access.

R_{BRSMZ} Software can use the MSR (immediate) instructions to directly write to PSTATE.{D, A, I, F, SP, PAN, UAO, DIT, SSBS}.

R_{CWYCH} The following PSTATE fields can be accessed at EL0:

- PSTATE.{N, Z, C, V, SSBS, DIT}.
- If SCTLR_EL1.UMA is 1, PSTATE.{D, A, I, F}.
- All other PSTATE access instructions can be executed at EL1 or higher and are UNDEFINED at EL0.

R_{HXYGT} Writes to the PSTATE fields have side-effects on various aspects of the PE operation. For side-effects caused by writes to a PSTATE field, all of the following are true:

- The side-effect is guaranteed not to be visible to earlier instructions in the Execution stream.
- The side-effect is guaranteed to be visible to later instructions in the Execution stream.

Other side-effects might occur but are not guaranteed.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D1.5 Resets and power domains

I_{DQXXZ} The PE logic is split into a Debug power domain and a Core power domain. Cold and Warm resets, reset elements in the Core power domain. An External debug reset can reset the Debug power domain. Other resets might be implemented.

D1.5.1 Power domains and reset domains

R_{PQLRX} The architecture defines the following power domains:

- The Core power domain.
- The Debug power domain.
- The system counter power domain.
- If trace is implemented, a power domain for the Trace Unit.
- If FEAT_RAS is implemented, there may be one or more IMPLEMENTATION DEFINED error recovery power domains. The relationship between these power domains and other power domains is IMPLEMENTATION DEFINED. For error records associated with the PE, this is the same as the Core power domain.

I_{QBPTS} The power domains are described as logical because the architecture does not require two physical power domains.

R_{PDMKB} It is IMPLEMENTATION DEFINED whether each power domain powers up and powers down separately or together.

R_{ZDVQQ} The Core power domain contains:

- Non-debug registers and logic.
- Self-hosted debug registers and logic.
- Shared debug registers and logic.
- Some external debug registers and logic.

R_{GTVLN} The Debug power domain contains:

- The interface between the PE and the external debugger.
- Some external debug registers.

R_{MLXWM} If an external debugger is connected to the PE, the Debug power domain is powered up.

I_{KTSWK} For more information on access permissions for the external debug interface registers, see Access permissions for the External debug interface registers or the individual register descriptions.

D1.5.2 Reset types

I_{YHXXB} The architecture defines the following resets:

- Warm reset.
- Cold reset.
- External Debug reset.
- Trace unit reset. See Resetting the trace unit.
- Timer reset.
- If FEAT_RAS is implemented, there may be one or more IMPLEMENTATION DEFINED error recovery resets. The relationship between this reset and other resets is IMPLEMENTATION DEFINED. For error records associated with the PE, the error recovery reset is the same as Warm reset.

I_{FMTCY} Other resets are IMPLEMENTATION DEFINED and can be mapped onto the architecturally-defined resets.

R_{PXXYQ} Mechanisms to assert resets, other than RMR_ELx, are IMPLEMENTATION DEFINED. One such mechanism is EDPRCR.CWRR. Any reset-asserting mechanism that software can command, including hardware mechanisms directly exposed to software, must only be accessible at the highest Exception level.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{HLDGD} When a Cold reset is asserted, for each register in the Core power domain, the reset sets the register to its architecturally-defined reset value.

R_{PJXVL} When a Warm reset is asserted, registers and logic in the Core power domain that are affected by the Warm reset are reset to their architecturally-defined reset value. Registers and logic that are only reset by a Cold reset are unaffected by a Warm reset.

R_{NBKHK} When an External Debug reset is asserted, the External Debug reset domain is reset and affected registers are set to their architecturally-defined External debug reset values. The Cold reset domain is unaffected.

R_{RKSZM} When a Cold reset is asserted, a Warm reset is asserted.

I_{VBZBW} Every register field that has storage belongs to exactly one reset domain. Each System register field definition defines the reset domain and the reset value.

I_{MWTXK} If RMR_ELx is implemented, writing 1 to RMR_ELx.RR requests a Warm reset.

R_{SGXSW} Writing 1 to RMR_ELx.RR is only a request for a Warm reset. The reset is not guaranteed to occur unless the following code sequence is executed:

```
; In addition, interrupts and debug requests for this PE should be disabled
; in the system before running this sequence to ensure the WFI suspends execution,
        MOV Wy, #3          ; for AArch64, #2 for AArch32; y is any register
        DSB                 ; ensure all stores etc are complete
        MSR RMR_ELx, Wy     ; request the reset
        ISB                 ; synchronize change to the RMR
Loop
        WFI                 ; enter a quiescent state
        B Loop
```

R_{DDTVZ} It is IMPLEMENTATION DEFINED whether a Warm reset can be asserted without using RMR_ELx.

I_{LWNDC} Arm recommends that a Warm reset can be asserted independently of a Cold reset.

I_{QLZGF} A Warm reset allows debugging across a reset of the PE logic in the Core power domain.

R_{SKGMH} It is IMPLEMENTATION DEFINED whether an External Debug reset and a Cold reset can be asserted independently.

I_{WTJYS} Arm recommends that when separate Core and Debug power domains are implemented, the External Debug reset and Cold reset can be asserted independently.

D1.5.3 Reset behavior

I_{TWYCF} Immediately after a reset, much of the PE state is architecturally UNKNOWN. However, some of the PE state is defined, see the individual register descriptions for more information. The state that is reset to known values is sufficient to permit predictable initial execution at the highest Exception level, such that this execution is then capable of initializing the remaining state of the system where necessary before use.

R_{JYLQV} When a Cold or Warm reset is asserted, all of the following occur:

- The PE enters the highest implemented Exception level.
- The stack pointer for the highest implemented Exception level, SP_ELx, is selected.

R_{BPKQV} When a Cold or Warm reset is deasserted, execution starts at an IMPLEMENTATION DEFINED address, anywhere in the *Physical Address* (PA) range. RVBAR_EL2, holds the address at which the PE starts execution.

I_{QSCPY} When a Cold or Warm reset is deasserted, the IMPLEMENTATION DEFINED address from which the PE starts execution is typically set either by a hardwired configuration of the PE or by configuration input signals.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{RRJLM} When a Warm reset is asserted, all of the following are architecturally UNKNOWN:

- All general-purpose, SIMD, and floating-point registers.
- The global exclusive monitor for the PE.
- The local exclusive monitor for the PE.
- System and Special-purpose registers with a reset value of architecturally UNKNOWN.

I_{KWHDG} When a Cold or Warm reset is asserted, PSTATE.{SS, IL, nRW, EL, SP, D, A, I, F, DIT} fields are set to known values.

R_{BGSHC} When a Cold or Warm reset is asserted, the TLBs and caches are in an IMPLEMENTATION DEFINED state.

R_{XNNNH} When a Cold or Warm reset has been asserted and before the memory management system is enabled, the TLBs, caches or both might need to be invalidated by IMPLEMENTATION DEFINED invalidation sequences.

R_{TZHFZ} When a Cold or Warm reset has been asserted and before Normal memory accesses are permitted to be Cacheable, the TLBs, caches or both might need to be invalidated by IMPLEMENTATION DEFINED invalidation sequences.

R_{KMCGN} If IMPLEMENTATION DEFINED resets are implemented, each IMPLEMENTATION DEFINED reset can treat the cache and TLB state differently.

R_{TJGFQ} If IMPLEMENTATION DEFINED resets are implemented, for each IMPLEMENTATION DEFINED reset, the TLBs, caches, or both might require a different IMPLEMENTATION DEFINED invalidation sequence.

I_{PHDMZ} The IMPLEMENTATION DEFINED invalidation sequence might be a NOP.

D1.5.3.1 External debug access to registers in reset

R_{VFMKQ} If External Debug reset is asserted, for each register that can be accessed by the external debug interface, a register access by the external debug interface has one of the following IMPLEMENTATION DEFINED results:

- The external debug interface can access the register, and the access result is IMPLEMENTATION DEFINED.
- The external debug interface cannot access the register.

I_{QHHMQ} If either a Cold or Warm reset is asserted, accesses of registers or register fields might be direct accesses or indirect side-effects of an access.

R_{YKRBF} If either a Cold or Warm reset is asserted, external debug interface accesses of a register or register field have the following effects:

Access	Register or register field reset behavior	Effect on register or register field
Write	Reset by the reset signal	Set to CONSTRAINED UNPREDICTABLE choice of the reset value or the value written.
Write	Not reset by the reset signal	Set to the value written.
Read	Reset by the reset signal	Returns an UNKNOWN value.
Read	Not reset by the reset signal	Returns the value of the register or register field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D1.6 Mechanisms for entering a low-power state

I_{RTNZD} The architecture provides the following mechanisms that software can use to indicate that the PE can enter low-power state:

- Wait for Event.
- Wait for Interrupt.

D1.6.1 Wait for Event

I_{ZZQGW} The Wait for Event mechanism behavior depends on the interaction of all of the following:

- The Event Register for the PE. See The Event Register.
- The Wait for Event (WFE) or Wait for Event with Timeout (WFET) instruction. See The Wait for Event and Wait for Event with Timeout instructions.
- WFE wakeup events. See WFE wakeup events in AArch64 state.
- The Send Event instructions, SEV and SEVL, that can cause WFE wakeup events. See The Send Event instructions.

R_{TJTFC} The architecture does not define the exact nature of the low-power state entered by WFE or WFET, except that when a WFE or WFET instruction is executed, memory coherency and architectural state are not lost.

I_{WBMQQ} The Wait for Event mechanism is associated with suspending execution on a PE for power saving, therefore Arm recommends that the Event Register is set infrequently. Software must only use the setting of the Event Register as a hint, and must not assume that any particular message is sent as a result of the setting of the Event Register.

D1.6.1.1 The Event Register

R_{PVKKP} The Event Register is a single bit register for each PE.

I_{TZVVM} The event register is a conceptual register and cannot be explicitly accessed.

I_{PDJMB} If the Event Register is 1, then an event has occurred since the register was last cleared and the event might require some action by the PE.

R_{RJYVC} When the PE executes a WFE or WFET instruction, all of the following apply:

- If the Event Register is 0, then the PE can enter the low-power state.
- If the Event Register is 1, then all of the following apply:
  - The PE does not suspend operation.
  - The Event Register is cleared to 0.
  - The WFE or WFET instruction completes immediately.

R_{DDGLZ} The reset value of the Event Register is UNKNOWN.

R_{XRZRK} The Event Register for a PE is set by any of the following:

- A Send Event instruction, SEV, executed by any PE in the system.
- A Send Event Local instruction, SEVL, executed by the PE.
- An exception return.
- The clearing of the global monitor for the PE.
- An event from a Generic Timer event stream, see Event streams.
- An event sent by some IMPLEMENTATION DEFINED mechanism.

I_{PJZBB} The Event Register is cleared only by a Wait for Event (WFE) instruction or a Wait for Event with Timeout (WFET) instruction.

I_{ZXJPD} Software cannot read or write the value of the Event Register directly.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D1.6.1.2 Trapping of WFE and WFET

I_{KMHTB} The WFE and WFET instructions are available at all Exception levels. Attempts to enter a low-power state made by software executing at EL0 or EL1 can be configured to trap to EL2.

D1.6.1.3 WFE wakeup events in AArch64 state

I_{LRPVP} If a WFE or WFET instruction puts a PE into low-power state, a WFE wakeup event received by the PE causes that PE to exit low-power state.

R_{KBMFC} All of the following are WFE wakeup events:

- The execution of an SEV instruction on any PE in a multiprocessor system.
- Any physical SError exception, physical IRQ interrupt, or physical FIQ interrupt received by the PE that is not disabled by EDSCR.INTdis and either of the following is true:
  - The interrupt is marked as A in the tables in Asynchronous exception masking, regardless of the value of the corresponding PSTATE.{A, I, F} mask bit.
  - The interrupt is marked as B in the tables in Asynchronous exception masking and any of the following are true:
    - The value of the corresponding PSTATE.{A, I, F} mask bit is 0.
    - AllIntMask is 0 and the IRQ or FIQ interrupt has Superpriority.
      See R_{MHWBP} and R_{GFXKY}.
- If all of the following apply, any virtual SError exception, virtual IRQ interrupt, or virtual FIQ interrupt received by the PE:
  - The PE is executing at EL1 or EL0.
  - The interrupt is not disabled by EDSCR.INTdis.
  - The interrupt or exception is marked as B in the table in Virtual interrupt masking and any of the following are true:
    - The value of the corresponding PSTATE.{A, I, F} mask bit is 0.
    - AllIntMask is 0 and the vIRQ or vFIQ interrupt has Superpriority.
      See R_{SNLJH} and R_{XSPSG}.
- If all the following apply, any delegated SError exception received by the PE:
  - The PE is executing at EL2, EL1, or EL0.
  - The exception is not disabled by EDSCR.INTdis.
  - Either of the following is true:
    - The exception is marked as A in the table in Delegated SError exception masking.
    - The exception is marked as B in the table in Delegated SError exception masking and PSTATE.A is 0.
- If halting is allowed, an asynchronous External Debug Request debug event. For the definition of halting, see Halting allowed and halting prohibited and External Debug Request debug event.
- An event sent by the timer event stream for the PE. See Event streams.
- An event caused by the clearing of the global monitor for the PE.
- An event sent by some IMPLEMENTATION DEFINED mechanism.
- When FEAT_WFxT is implemented, for `WFET` instructions, a local timeout event caused by the virtual count threshold value, expressed in CNTVCT_EL0, being equaled or exceeded.

I_{FRZZX} Not all of the wakeup events in R_{KBMFC} set the Event Register.

D1.6.1.4 The Send Event instructions

R_{DQRMZ} The Send Event instructions are:

- `SEV`, Send Event: Causes an event to be signaled to all PEs in a multiprocessor system.
- `SEVL`, Send Event Local: Causes an event to be signaled locally without requiring the event to be signaled to other PEs in a multiprocessor system.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{FVQBK} An SEVL instruction might signal an event to other PEs by an IMPLEMENTATION DEFINED mechanism, but it is not required to do so.

R_{HTDFW} The mechanism that signals an event to other PEs is IMPLEMENTATION DEFINED.

R_{RLHHM} When an event is signaled by an SEV instruction, the ordering of the event with respect to the completion of memory accesses by instructions before the SEV instruction is not guaranteed.

I_{DVZXD} Arm recommends that software includes a DSB instruction before any SEV instruction. DSB instruction ensures that no instructions, including any SEV instructions, that appear in program order after the DSB instruction, can execute until the DSB instruction has completed. See Data Synchronization Barrier.

R_{QJDQV} The SEVL instruction appears to execute in program order relative to any subsequent WFE or WFET instruction executed on the same PE.

I_{HWHBP} The behavior in R_{QJDQV} applies without the need for any explicit insertion of barrier instructions.

D1.6.2 Wait for Interrupt mechanism

I_{LMJGQ} A PE can use the Wait for Interrupt mechanism to enter a low-power state.

R_{DBXLJ} Software can use the Wait for Interrupt (WFI) instruction to cause the PE to enter a low-power state.

R_{HSPJD} If a Wait for Interrupt (WFI) instruction causes the PE to enter a low-power state, the PE remains in that low-power state until any of the following occur:

- The PE detects a WFI wakeup event.
- An IMPLEMENTATION DEFINED wake event that is architecturally permitted to occur at any time.
- A reset is asserted.

I_{MFLXH} The architecture permits all of the following:

- The PE is permitted to leave the low-power state for any reason.
- The PE is permitted to treat WFI as a NOP, but this is not recommended for lowest power operation.

R_{MRVLC} When the PE leaves a low-power state that was entered because of a WFI instruction, the WFI instruction completes.

R_{HJPJY} Except for all or the following, the architecture does not define the exact nature of the low-power state:

- When a WFI instruction is executed, the architecture requires that memory coherency is not lost.
- If the system is configured such that the WFI instruction can be completed, then the architecture requires that the architectural state is not lost.

I_{WWYXS} In some implementations, based on the configuration of system specific registers, WFI can be used as part of a powerdown sequence where no interrupts will cause WFI wakeup events, and restoration of power involves resetting of the PE. In those cases, the WFI is permitted to cause a loss of architectural state, as it is assumed that this state will have been saved by software as part of the powerdown sequence before the WFI.

I_{CHMWP} The WFI instruction are available at all Exception levels. Attempts to enter a low-power state made by software executing at EL0, EL1, or EL2 can be configured to trap to a higher Exception level.

D1.6.2.1 WFI wakeup events

I_{CYKVZ} In this section, AllIntMask refers to the value described in R_{XZPDT}.

R_{VRLPB} All of the following are WFI wakeup events:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Any physical SError exception, physical IRQ interrupt, or physical FIQ interrupt received by the PE that is marked as A or B in the tables in Asynchronous exception masking, regardless of the value of the corresponding PSTATE.{A, I, F} mask bit.
- If the PE is executing at EL1 or EL0, any virtual SError exception, virtual IRQ interrupt, or virtual FIQ interrupt received by the PE, that is marked as B in the tables in Virtual interrupt masking, regardless of the value of the corresponding PSTATE.{A, I, F} mask bit.
- If the PE is executing at EL2, EL1, or EL0, any delegated SError exception received by the PE that is marked as A or B in the table in Delegated SError exception masking, regardless of the value of PSTATE.A.
- If halting is allowed, an asynchronous External Debug Request debug event. For the definition of halting, see Halting allowed and halting prohibited and External Debug Request debug event.
- An event sent by an IMPLEMENTATION DEFINED mechanism.

R_{NLVGY} WFI wakeup events are never disabled by EDSCR.INTdis, and are never masked by the PSTATE.{A, I, F} mask bits, or by the state of AllIntMask. If wakeup is invoked by an interrupt that is disabled or masked, the interrupt is not taken.

I_{VCDGW} Because debug events are WFI wakeup events, Arm recommends that Wait for Interrupt is used as part of an idle loop rather than waiting for a single specific interrupt event to occur and then moving forward. This ensures that the intervention of debug while waiting does not significantly change the function of the program being debugged.

I_{FXRXC} If the PE is in WFx state when a PMU Capture event occurs, it must wake-up to service the request. Arm recommends the Capture event is not treated as an IMPLEMENTATION DEFINED WFx wake-up event and the PE returns to standby after. See PMU snapshots.

I_{ZQWXZ} The architecture does not require the WFI mechanism to drain down any pending memory activity before suspending execution, and software must not rely on the WFI mechanism operating in this way.

D1.6.3 Pending PMU Exception

I_{WNMSV} A pending PMU exception is not required to be a WFE wakeup event or WFI wakeup event.

Arm recommends that an unmasked asynchronous PMU exception that is pending when a `WFE` or `WFI` instruction is executed is treated as an IMPLEMENTATION DEFINED wakeup event, to allow timely processing of the PMU event. For example, when other telemetry data needs to be collected by software from the system.

However, if a PMU overflow occurs as the PE enters a low-power WFx state, then the PE might not register the pending PMU exception before entering the WFx state.

A similar condition applies when using PMU interrupts, because the PE has to assert the interrupt request to the interrupt controller before entering the WFx state for it to be a wakeup event. This is due to IMPLEMENTATION SPECIFIC timings that determine whether a PMU overflow causes a WFx wakeup or prevents the PE from entering the WFx state.

I_{VPTQY} In I_{WNMSV}, pending means that, if the PMU exception were unmasked, a PMU exception would be taken by a simple sequential execution of the program before the execution of the next instruction.

D1.6.4 Using WFI to indicate an idle state on bus interfaces

R_{TZPSV} When all of the following are true, the IMPLEMENTATION DEFINED mechanism for entering powerdown state can be completed:

- The IMPLEMENTATION DEFINED mechanism forces quiescence on a PE.
- The IMPLEMENTATION DEFINED mechanism prevents any possible WFI wakeup events.

R_{VWDWJ} The mechanism for entering powerdown state is IMPLEMENTATION DEFINED.

I_{FFGGG} Although the mechanism for entering powerdown state is IMPLEMENTATION DEFINED, the WFI and WFE mechanisms are often used to enter powerdown state.

R_{RMBKR} If a WFI powerdown mechanism is implemented, when a WFI instruction is executed, the PE completes all current operations and any associated bus activity also completes. When all current PE operations and associated bus activity are complete, the PE can signal to an external power controller that there is no ongoing bus activity.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{CLTZY} After a WFI instruction is executed and all currently executing instructions and related bus activity has completed, the PE is waiting for an interrupt and is in *idle state*. If in idle state, the PE can process memory-mapped or external debug interface accesses to debug registers.

I_{CYBLG} The indication of idle state to the system normally only applies to the non-debug functional interfaces used by the PE, not the debug interfaces.

R_{DHHKY} If OSDLR_EL1.DLK is 1, the PE does not signal the idle state to the control logic unless it can also guarantee that the debug interface is idle. For more information about the OS Double Lock, see Debug behavior when the OS Double Lock is locked.

I_{WRZGC} In a PE that implements separate Core and Debug power domains, the debug interface referred to in this section is the interface between the Core and Debug power domains, since the signal to the power controller indicates that the Core power domain is idle. For more information about the power domains, see Power domains and debug.

I_{RFLDH} The exact nature of the debug interface is IMPLEMENTATION DEFINED, but the use of Wait for Interrupt as the only architecturally-defined mechanism that completely suspends execution makes it suitable as the preferred powerdown entry mechanism.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D1.7 Self-hosted debug

I_{MJGPY} The architecture supports both of the following:

Self-hosted debug

The PE itself hosts a debugger. The debugger programs the PE to generate debug exceptions. Debug exceptions are accommodated in the Arm Exception model.

External debug

The PE is controlled by an external debugger. The debugger programs the PE to generate debug events, that cause the PE to enter Debug state. In Debug state, the PE is halted.

D1.7.1 Debug exceptions

I_{PWPHY} If a debugger has programmed the PE to generate Debug exceptions, the Debug exceptions occur during normal program flow.

I_{DSWBH} For example, a software developer might use a debugger contained in an operating system to debug an application. To do this, the debugger might enable one or more debug exceptions.

I_{BXVCF} The debug exceptions are all of the following:

- Breakpoint Instruction exceptions.
- Breakpoint exceptions.
- Watchpoint exceptions.
- Vector Catch exceptions.
- Software Step exceptions.

I_{VYKHK} The PE can only generate a particular debug exception if all of the following are true:

- Debug exceptions are enabled from the current Exception level and Security state. See Enabling debug exceptions from the current Exception level and Security state. Breakpoint Instruction exceptions are always enabled from the current Exception level and Security state.
- A debugger has enabled that particular debug exception. All of the debug exceptions except for Breakpoint Instruction exceptions have an enable control contained in the MDSCR_EL1. See The debug exception enable controls.

I_{GRWSG} Breakpoints and watchpoints can cause entry to Debug state instead of causing debug exceptions. See About External Debug.

D1.7.2 The PSTATE debug mask bit, D

R_{DYRLS} The debug exception mask bit, PSTATE.D, can mask Watchpoint, Breakpoint, and Software Step exceptions if the target Exception level is the current Exception level.

R_{RDYDM} If in AArch64 state and the target Exception level of a debug exception is lower than the current Exception level, the debug exception is masked.

R_{JWZFQ} If the target Exception level is higher than the current Exception level, debug exceptions are not masked by PSTATE.D.

R_{KCHGM} If PSTATE.D is 1 and the PE is already handling an exception in AArch64 state, the Watchpoint, Breakpoint, and Software Step exceptions are masked.

I_{LVYYH} If an exception is already being handled, masking prevents software from generating another instance of an exception and prevents recursive entry into the exception handler and loss of return state.

R_{NCHXS} Masking debug interrupts with PSTATE.D prevents the generation of new debug exceptions, therefore, any masked debug exceptions are not taken if PSTATE.D is later set to 0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{VSTYD} The behavior described in R_{NCHXS} differs from the behavior for interrupts, where the PSTATE.{A, I, F} mask prevents the interrupt from being taken, but instead the interrupt remains pending.

I_{KWFMZ} When an exception is taken, PSTATE.D is set to 1.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D1.8 Event monitors

I_{YVDTB} The architecture supports the following non-invasive architectural components that allow for event monitoring:

Performance Monitors

The Performance Monitors have a wide feature set, flexible selection of counted events, and are read/write in operation.

D1.8.1 The Performance Monitors Extension

I_{ZSPVV} The System registers provide access to a Performance Monitoring Unit (PMU), defined as the OPTIONAL Performance Monitors Extension to the architecture, a non-invasive debug resource that provides information about the operation of the PE.

The PMU provides:

- A 64-bit cycle counter.
- An IMPLEMENTATION DEFINED number of event counters. The events that can be counted are:
  - Architectural and microarchitectural events that are likely to be consistent across many microarchitectures. The PMU architecture uses event numbers to identify an event, and the PMU specification defines which event number must be used for each of these architectural and microarchitectural events.
  - IMPLEMENTATION SPECIFIC events. The PMU specification reserves event numbers for IMPLEMENTATION SPECIFIC events.

For more information, see The Performance Monitors Extension.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter D2
AArch64 Self-hosted Debug

When the PE is using self-hosted debug, it generates *debug exceptions*. This chapter describes the AArch64 self-hosted debug Exception model. It is organized as follows:

Introductory information

- About self-hosted debug.
- The debug exception enable controls.

The debug Exception model

- Routing debug exceptions.
- Enabling debug exceptions from the current Exception level and Security state.
- The effect of powerdown on debug exceptions.
- Summary of the routing and enabling of debug exceptions.
- Pseudocode description of debug exceptions.

The debug exceptions

- Breakpoint Instruction exceptions.
- Breakpoint exceptions.
- Watchpoint exceptions.
- Software Step exceptions.

Synchronization requirements

The behavior of self-hosted debug after changes to System registers, or after changes to the authentication interface, but before a Context Synchronization event guarantees the effects of the changes:

- Synchronization and debug exceptions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D2.1 About self-hosted debug

Self-hosted debug supports debugging through the generation and handling of *debug exceptions*, which are taken using the Exception model described in The AArch64 System Level Programmers' Model. This section introduces some terms that are used in describing self-hosted debug, and then introduces the debug exceptions. See:

- Definition of a debugger in the context of self-hosted debug.
- Context ID and Process ID.
- About debug exceptions.

D2.1.1 Definition of a debugger in the context of self-hosted debug

Within this chapter, *debugger* means that part of an operating system, or higher level of system software, that handles debug exceptions and programs the Debug system registers. An operating system with rich application environments might provide debug services that support a debugger user interface executing at EL0. From the architectural perspective, the debug services are the debugger.

D2.1.2 Context ID and Process ID

A CONTEXTIDR_ELx identifies the current *Context ID*, which is used by:

- The debug logic, for breakpoint and watchpoint matching.
- Implemented trace logic, to identify the current process.

In AArch64 state, the CONTEXTIDR_ELx has a single field, PROCID, that is defined as the *Process Identifier* (Process ID). Therefore, in AArch64 state, the Context ID and Process ID are identical.

D2.1.3 About debug exceptions

Debug exceptions occur during normal program flow if a debugger has programmed the PE to generate them. For example, a software developer might use a debugger contained in an operating system to debug an application. To do this, the debugger enables one or more debug exceptions. The debug exceptions that can be generated in stage 1 of an AArch64 translation regime are:

- Breakpoint Instruction exceptions.
- Breakpoint exceptions, generated by hardware breakpoints.
- Watchpoint exceptions, generated by hardware watchpoints.
- Software Step exceptions.

Other than Breakpoint Instruction exceptions, the PE can generate a particular debug exception only if all of the following are true:

- The OS Lock is unlocked.
- `DoubleLockStatus()` == FALSE.
- Debug exceptions are not disabled from the current Exception level and Security state.
  See Enabling debug exceptions from the current Exception level and Security state.
- A debugger has not disabled that particular debug exception.
  All of the debug exceptions except for Breakpoint Instruction exceptions have an enable control contained in the MDSCR_EL1. See The debug exception enable controls.

Breakpoint Instruction exceptions are always enabled.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

If *halting is allowed* and EDSCR.HDE is 1, hardware breakpoints and watchpoints cause entry to Debug state instead of causing debug exceptions. In Debug state, the PE is halted.
For the definition of halting is allowed, see Halting allowed and halting prohibited.

The following list summarizes each of the debug exceptions:

Breakpoint Instruction exceptions

Breakpoint instructions generate these. Breakpoint instructions are instructions that software developers can use to cause exceptions at particular points in the program flow.

The breakpoint instruction is `BRK #<immediate>`. Whenever one of these is committed for execution, the PE takes a Breakpoint Instruction exception.

Breakpoint Instruction exceptions cannot be masked. The PE takes Breakpoint Instruction exceptions regardless of both of the following:

- The current Exception level.
- The current Security state.

For more information, see Breakpoint Instruction exceptions.

Breakpoint exceptions

The architecture provides 2-16 hardware breakpoints. These can be programmed to generate Breakpoint exceptions based on one or more of particular instruction addresses, accesses to any address in an instruction address range, or particular PE contexts.

For example, a software developer might program a hardware breakpoint to generate a Breakpoint exception whenever the instruction with address `0x1000` is committed for execution.

The architecture supports the following types of hardware breakpoint for use in stage 1 of an AArch64 translation regime:

- Address.
  - Comparisons are made with the virtual address of each instruction in the program flow.
- Context:
  - Context ID Match. Matches with the Context ID held in the CONTEXTIDR_EL1, and if FEAT_VHE is implemented, CONTEXTIDR_EL2.

An Address breakpoint can link to a Context-matching breakpoint, so that the Address breakpoint generates a Breakpoint exception only if the PE is in a particular context when the address match occurs.

A breakpoint generates a Breakpoint exception whenever an instruction that causes a match is committed for execution.

If halting is allowed and EDSCR.HDE is 1, hardware breakpoints cause entry to Debug state. That is, they halt the PE. See Debug State.

Otherwise:

- If debug exceptions are enabled, hardware breakpoints cause Breakpoint exceptions.
- If debug exceptions are disabled, hardware breakpoints are ignored.

For more information, see Breakpoint exceptions.

Watchpoint exceptions

The architecture provides 2-16 hardware watchpoints. These can be programmed to generate Watchpoint exceptions based on accesses to particular data addresses, or based on accesses to any address in a data address range.

For example, a software developer might program a hardware watchpoint to generate a Watchpoint exception on an access to any address in the data address range `0x1000` - `0x101F`.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A hardware watchpoint can link to a hardware breakpoint if the hardware breakpoint is a Context matching breakpoint with linking enabled. In this case, the watchpoint generates a Watchpoint exception only if the PE is in a particular context when the data address match occurs.

The smallest data address size that a watchpoint can be programmed to match on is a byte. A single watchpoint can be programmed to match on one or more bytes.

A watchpoint generates a Watchpoint exception whenever an instruction that initiates an access that causes a match is committed for execution.

If halting is allowed and EDSCR.HDE is 1, hardware watchpoints cause entry to Debug state. That is, they halt the PE. See Debug State.

Otherwise:

- If debug exceptions are enabled, hardware watchpoints cause Watchpoint exceptions.
- If debug exceptions are disabled, hardware watchpoints are ignored.

For more information, see Watchpoint exceptions.

Software Step exceptions

Software step is a resource that a debugger can use to make the PE single-step instructions.

For example, by using software step, debugger software executing at a higher Exception level can debug software executing at a lower Exception level, by making it single-step instructions.

After the software being debugged has single-stepped an instruction, the PE takes a Software Step exception.

If debug exceptions are enabled, Software Step exceptions can be generated.

If debug exceptions are disabled, software step is inactive.

For more information, see Software Step exceptions.

Table D2-1 summarizes PE behavior and shows the location of the pseudocode for each of the debug exceptions.

Table D2-1 PE behavior and pseudocode for each of the debug exceptions

Debug exception	PE behavior if debug exceptions are:		Pseudocode
	Enabled	Disabled	
Breakpoint Instruction exceptions	Takes the exception	Takes the exception	Pseudocode description of Breakpoint Instruction exceptions
Breakpoint exceptions	Takes the exception[a]	Ignored	Pseudocode description of Breakpoint exceptions taken from an AArch64 stage 1 translation regime
Watchpoint exceptions	Takes the exception[a]	Ignored	Pseudocode description of Watchpoint exceptions taken from AArch64 state
Software Step exceptions	Takes the exception	Not applicable[b]	Pseudocode description of Software Step exceptions

a. If halting is allowed and EDSCR.HDE is 1, hardware breakpoints and watchpoints cause the PE to enter Debug state instead of causing debug exceptions. See Debug State.

b. Software Step is inactive if debug exceptions are disabled. No Software Step exceptions can be generated.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D2.2 Routing debug exceptions

Debug exceptions are routed to the *debug target Exception level*, EL_D, according to the following controls:

- HCR_EL2.TGE.
- MDCR_EL2.TDE.
- The Security state when the exception is taken.
- The Exception level where the exception is taken.

This means that, for all debug exceptions, EL_D is either EL1 or EL2.

Note

If EL2 is not implemented, the Effective value of HCR_EL2.TGE is 0 and the Effective value of MDCR_EL2.TDE is 0. Throughout this section, references to the values of these fields are to the Effective values of the fields.

The routing of debug exceptions is as follows:

Debug exceptions taken when EL2 is implemented and enabled in the current Security state

The routing of debug exceptions taken depends on the values of MDCR_EL2.TDE and HCR_EL2.TGE:

If the Effective value of {MDCR_EL2.TDE, HCR_EL2.TGE} is not {0, 0}

Debug exceptions are routed to EL2, EL_D is EL2.

Otherwise

Debug exceptions behave as follows:

- Debug exceptions taken from EL1 and EL0 are routed to EL1. EL_D is EL1.
- Breakpoint Instruction exceptions taken from EL2 are routed to EL2.
- All other debug exceptions are disabled from EL2 using AArch64.

TDE or TGE Means the logical OR of the Effective value of MDCR_EL2.TDE and the Effective value of HCR_EL2.TGE.

Table D2-2 Routing of exceptions

TDE or TGE	EL_D when executing in:		
	EL0	EL1	EL2
0	EL1	EL1	(EL1)
1	EL2	EL2	EL2

D2.2.1 Pseudocode description of routing debug exceptions

`DebugTarget()` returns the current debug target Exception level.

`DebugTargetFrom()` returns the debug target Exception level for the specified Security state.

These functions are described in R-profile Architecture Pseudocode.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D2.3 The debug exception enable controls

The enable controls for each debug exception are as follows:

Breakpoint Instruction exceptions

None. Breakpoint Instruction exceptions are always enabled.

Breakpoint exceptions

MDSCR_EL1.MDE, plus an enable control for each breakpoint, DBGBCR<n>_EL1.E.

Watchpoint exceptions

MDSCR_EL1.MDE, plus an enable control for each watchpoint, DBGWCR<n>_EL1.E.

Vector Catch exceptions

MDSCR_EL1.MDE.Vector catch exceptions cannot be generated in AArch 64 state.

Software Step exceptions

MDSCR_EL1.SS.

In addition, for all debug exceptions other than Breakpoint Instruction exceptions, software must configure the controls that enable debug exceptions from the current Exception level and Security state. See Enabling debug exceptions from the current Exception level and Security state.

The PE cannot take a debug exception if debug exceptions are disabled from either the current Exception level or the current Security state.

Breakpoint Instruction exceptions are always enabled from the current Exception level and Security state.

D2.3.1 Enabling debug exceptions from the current Exception level and Security state

Debug exceptions are enabled from the current Exception level, where EL_D is the Exception level defined in Routing debug exceptions, as follows:

- When executing at any Exception level that is higher than EL_D:
  - Breakpoint Instruction exceptions are enabled.
  - All other debug exceptions are disabled.
- When executing at EL_D:
  - Breakpoint Instruction exceptions are enabled.
  - All other debug exceptions are disabled if either of the following is true:
    - The Local (kernel) Debug Enable bit, MDSCR_EL1.KDE, is 0.
    - The Debug exception mask bit, PSTATE.D, is 1.

  This means that a debugger must explicitly enable these debug exceptions from EL_D by setting MDSCR_EL1.KDE to 1 and PSTATE.D to 0.
- When executing at EL1, EL_D is EL2, and FEAT_NV2 is implemented:
  - Watchpoint debug exceptions generated by a System register access converted to a memory access because the Effective value of HCR_EL2.NV2 is 1 are disabled if MDSCR_EL1.KDE is 0 and enabled if MDSCR_EL1.KDE is 1. The value of PSTATE.D is ignored.
  - All other debug exceptions are enabled.
- Otherwise all debug exceptions are enabled.

D2.3.2 Pseudocode description of enabling debug exceptions

`AArch64.GenerateDebugExceptions()` determines whether debug exceptions other than Breakpoint Instruction exceptions are enabled from the current Exception level and Security state.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`AArch64.GenerateDebugExceptionsFrom()` determines whether debug exceptions other than Breakpoint Instruction exceptions are enabled from the specified Exception level and Security state.

These functions are described in R-profile Architecture Pseudocode.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D2.4 The effect of powerdown on debug exceptions

Debug OS Save and Restore sequences describes the *powerdown save routine* and the *restore routine*.

When executing either routine, software must use the OS Lock to disable generation of all of the following:

- Breakpoint exceptions.
- Watchpoint exceptions.
- Software Step exceptions.

This is because the generation of these exceptions depends on the state of the debug registers, and the state of the debug registers might be lost over these routines.

If the OS Lock is unlocked, and `DoubleLockStatus()` == FALSE, debug exceptions other than Breakpoint Instruction exceptions are enabled.

If OS Lock is locked, or if `DoubleLockStatus()` ==TRUE, debug exceptions other than Breakpoint Instruction exceptions are disabled.

Breakpoint Instruction exceptions are enabled regardless of the state of the OS Lock and the OS Double Lock.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D2.5 Summary of the routing and enabling of debug exceptions

Behavior is as follows:

Breakpoint Instruction exceptions

These are always enabled, regardless of the current Exception level and Security state. A Breakpoint Instruction exception taken from EL2 is routed to EL2. A Breakpoint Instruction exception taken from EL0 or EL1 is always routed to EL_D.

All other debug exceptions

Table D2-3 shows the valid combinations of MDCR_EL2.TDE, MDSCR_EL1.KDE, and PSTATE.D, and for each combination shows where these exceptions are enabled from and where they are taken to.

In the table:

Lock Means the value of (OSLSR_EL1.OSLK == '1' || `DoubleLockStatus()`).

TGE Means the value of HCR_EL2.TGE.

TDE Means the value of MDCR_EL2.TDE.

KDE Means the value of MDSCR_EL1.KDE.

D Means the value of PSTATE.D.

n/a Means not applicable. The PE cannot be executing at this Exception level.

- Means that debug exceptions are disabled from that Exception level.

Table D2-3 Routing of Breakpoint, Watchpoint, and Software Step exceptions

Debug state	Lock	TGE	TDE	KDE	D	EL_D when enabled from:		
						EL0	EL1	EL2
Yes	X	X	X	X	X	-	-	-
No	TRUE	X	X	X	X	-	-	-
	FALSE	X	X	X	X	-	-	-
		X	X	0	X	EL1	-	n/a
				1	0	EL1	EL1	n/a
					1	EL1	-	n/a
				1	0	EL1	EL1	-
					1	EL1	-	-
No	FALSE	0	1	0	X	EL2	EL2	-
				1	0	EL2	EL2	EL2
					1	EL2	EL2	-
		1	X	0	X	EL2	n/a	-
				1	0	EL2	n/a	EL2
					1	EL2	n/a	-
		0	0	0	X	EL1	-	-
				1	0	EL1	EL1	-
					1	EL1	-	-
			1	0	X	EL2	EL2	-

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Debug state	Lock	TGE	TDE	KDE	D	EL_D when enabled from:		
						EL0	EL1	EL2
				1	0	EL2	EL2	EL2
					1	EL2	EL2	-
		1	X	0	X	EL2	n/a	-
				1	0	EL2	n/a	EL2
					1	EL2	n/a	-

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D2.6 Pseudocode description of debug exceptions

The `AArch64.Abort()` function processes `FaultRecord` objects, as described in Abort exceptions, and generates a debug exception.

Some functions called by `AArch64.Abort()` are:

- `AArch64.BreakpointException()`.
- `AArch64.WatchpointException()`.
- `AArch64.InstructionAbort()`.
- `AArch64.DataAbort()`.

These functions are defined in R-profile Architecture Pseudocode.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D2.7 Breakpoint Instruction exceptions

This section describes Breakpoint Instruction exceptions.

A Breakpoint Instruction exception results from the execution of an A64 `BRK` instruction.

The following subsections describe Breakpoint Instruction exceptions in a translation regime:

- About Breakpoint Instruction exceptions.
- Breakpoint instructions.
- Exception syndrome information and preferred return address.
- Pseudocode description of Breakpoint Instruction exceptions.

D2.7.1 About Breakpoint Instruction exceptions

A *breakpoint* is an event that results from the execution of an instruction, which is based on either:

- The instruction address, the PE context, or both. This type of breakpoint is called a *hardware breakpoint.*
- The instruction itself. That is, the instruction is a *breakpoint instruction*. These can be included in the program that the PE executes. This type of breakpoint is called a *software breakpoint*.

Breakpoint Instruction exceptions, which this section describes, are software breakpoints. Breakpoint exceptions describes hardware breakpoints.

There is no enable control for Breakpoint Instruction exceptions. They are always enabled, and cannot be masked.

A Breakpoint Instruction exception is generated whenever a breakpoint instruction is committed for execution, regardless of the current Exception level.

Note

Debuggers using breakpoint instructions must be aware of the rules for concurrent modification and execution of instructions. See Concurrent modification and execution of instructions.

D2.7.2 Breakpoint instructions

The breakpoint instruction in the A64 instruction set is `BRK #<immediate>`. It is unconditional.

For details of the instruction encoding, see BRK.

D2.7.3 Exception syndrome information and preferred return address

See the following:

- Exception syndrome information.
- Preferred return address.

D2.7.3.1 Exception syndrome information

On taking a Breakpoint Instruction exception, the PE records information about the exception in the *Exception Syndrome Register* (ESR) at the Exception level the exception is taken to in ESR_ELx

shows the information that the PE records.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D2-4 Information recorded in the ESR_ELx

ESR_ELx field	Information recorded in ESR_EL1 or ESR_EL2	
Exception Class, EC	The PE sets this to `0x3C` for an A64 `BRK` instruction.	
Instruction Length, IL	The PE sets this to 1 for an A64 `BRK` instruction.	
Instruction Specific Syndrome, ISS	**ISS[24:16]**	RES0.
	ISS[15:0]	The PE copies the instruction Comment field value into here, zero extended as necessary.

Note

- If debug exceptions are routed to EL2, it is the exception that is routed, not the instruction that is trapped. Therefore, if a Breakpoint Instruction exception is routed to EL2, ESR_EL2.EC is set to the same value as if the exception was taken to EL1.
- For information about how debug exceptions can be routed to EL2, see Routing debug exceptions.

D2.7.3.2 Preferred return address

The preferred return address is the address of the breakpoint instruction, not the next instruction. This is different to the behavior of other exception-generating instructions, like `SVC`.

D2.7.4 Pseudocode description of Breakpoint Instruction exceptions

`AArch64.SoftwareBreakpoint()` generates a Breakpoint Instruction exception.

This function is defined in R-profile Architecture Pseudocode.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D2.8 Breakpoint exceptions

This section describes Breakpoint exceptions in stage 1 of an AArch64 translation regime.

This section contains the following subsections:

- About Breakpoint exceptions.
- Breakpoint types and linking of breakpoints.
- Execution conditions for which a breakpoint generates Breakpoint exceptions.
- Breakpoint instruction address comparisons.
- Breakpoint context comparisons.
- Breakpoint usage constraints.
- Preferred return address.
- Pseudocode description of Breakpoint exceptions taken from an AArch64 stage 1 translation regime.

D2.8.1 About Breakpoint exceptions

A *breakpoint* is an event that results from the execution of an instruction, which is based on either:

- The instruction address, the PE context, or both. This type of breakpoint is called a *hardware breakpoint.*
- The instruction itself. That is, the instruction is a *breakpoint instruction*. These can be included in the program that the PE executes. This type of breakpoint is called a *software breakpoint*.

Breakpoint exceptions are generated by *Breakpoint debug events*. Breakpoint debug events are generated by hardware breakpoints. Software breakpoints are described in Breakpoint Instruction exceptions.

An implementation can include between 2-16 hardware breakpoints.

To use an implemented hardware breakpoint, a debugger programs the following registers for the breakpoint:

- The *Breakpoint Control Register*, DBGBCR<n>_EL1. This contains controls for the breakpoint, for example an enable control.
- The *Breakpoint Value Register*, DBGBVR<n>_EL1. This holds the value used for breakpoint matching, which is one of:
  - An instruction virtual address.
  - A Context ID.
  - A VMID value.
  - A concatenation of both a Context ID value and a VMID value.
  - If FEAT_VHE is implemented, a concatenation of two Context ID values.

These registers are numbered, so that:

- DBGBCR0_EL1 and DBGBVR0_EL1 are for breakpoint number zero.
- DBGBCR1_EL1 and DBGBVR1_EL1 are for breakpoint number one.
- . . .
- DBGBCR<n-1>_EL1 and DBGBVR<n-1>_EL1 are for breakpoint number (n-1).

A debugger can link a breakpoint that is programmed with an address and a breakpoint that is programmed with anything other than an address together, so that a Breakpoint debug event is generated only if both breakpoints match.

For each instruction in the program flow, all of the breakpoints are tested. When a breakpoint is tested, it generates a Breakpoint debug event if all of the following are true:

- The breakpoint is enabled. That is, the Effective value of DBGBCR<n>_EL1.E is 1.
- The conditions specified in the DBGBCR<n>_EL1 are met.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The comparison with the value held in the DBGBVR<n>_EL1 is successful.
- If the breakpoint is linked to another breakpoint, the comparisons made by that other breakpoint are also successful.
- The instruction is committed for execution.

If halting is allowed and EDSCR.HDE is 1, Breakpoint debug events cause entry to Debug state.

Otherwise, if debug exceptions are:

- Enabled, Breakpoint debug events generate Breakpoint exceptions.
- Disabled, Breakpoint debug events are ignored.

Note

The remainder of this Breakpoint exceptions section, including all subsections, describes breakpoints as generating Breakpoint exceptions.
However, the behavior described also applies if breakpoints are causing entry to Debug state.

The debug exception enable controls describes the enable controls for Breakpoint debug events.

D2.8.2 Breakpoint types and linking of breakpoints

Each implemented breakpoint is one of the following:

- A *context-aware* breakpoint. This is a breakpoint that can be programmed to generate a Breakpoint exception on any one of the following:
  - An instruction address match or mismatch.
  - A Context ID match, with the value held in CONTEXTIDR_EL1 or the value held in CONTEXTIDR_EL2.
- A breakpoint that is not *context-aware*. These can only be programmed to generate a Breakpoint exception on an instruction address match.

Up to 16 *context-aware* breakpoints can be implemented, and the number implemented is identified by ID_AA64DFR0_EL1.CTX_CMPs.

The highest numbered breakpoints are *context-aware*.

At least one breakpoint must be *context-aware*.

Any breakpoint that is programmed to generate a Breakpoint exception on an instruction address match is categorized as an *Address breakpoint*. Breakpoints that are programmed to match on anything else are categorized as *Context breakpoints*.

When a debugger programs a breakpoint to be an Address or a Context breakpoint, it must also program that breakpoint so that it is either:

- Used in isolation. In this case, the breakpoint is called an *Unlinked breakpoint*.
- Linked to another Context matching breakpoint. In this case, the breakpoint is called a *Linked breakpoint*.
- Enabled for linking from another breakpoint or watchpoint. In this case, the breakpoint has *Linking enabled* and does not generate debug events in isolation.

By linking an Address breakpoint and a Context breakpoint together, the debugger can create a breakpoint pair that generates a Breakpoint exception only if the PE is in a particular context when an instruction address match occurs. This is shown in Example D2-1.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example D2-1 Linking an Address breakpoint and a Context breakpoint

For example, a debugger might:

1. Program breakpoint number one to be a *Linked Address Match breakpoint*.
2. Program breakpoint number five to be a *Context matching breakpoint with linking enabled*.
3. Link these two breakpoints together. A Breakpoint exception is generated only if both the instruction address matches and the Context ID matches.

The *Breakpoint Type* field for a breakpoint, DBGBCR<n>_EL1.BT, controls the breakpoint type and whether the breakpoint is enabled for linking.

D2.8.2.1 Rules for linking breakpoints

The rules for breakpoint linking are as follows:

- Only Linked Address breakpoint types can be linked.
- Any type of Linked Address breakpoint can link to any type of Context matching breakpoint with linking enabled.
  The *Linked Breakpoint Number* field, DBGBCR<n>_EL1.LBN, for the Linked Address breakpoint specifies the particular Context matching breakpoint that the Linked Address breakpoint links to, and:
  - The LBN field are ignored if the breakpoint is not a Linked Address breakpoint.
  - DBGBCR<n>_EL1.{SSC, HMC, PMC} for the Linked Address breakpoint define the execution conditions that the breakpoint pair generates Breakpoint exceptions for. See Execution conditions for which a breakpoint generates Breakpoint exceptions.
  - DBGBCR<n>_EL1.{SSC, HMC, PMC} for the Context matching breakpoint with linking enabled are ignored.
- Linked Address breakpoints cannot link to watchpoints. The LBN field always specify a breakpoint.
- If a Linked Address breakpoint links to a breakpoint that is not *context-aware* or does not have linking enabled, the behavior of the Linked Address breakpoint is CONSTRAINED UNPREDICTABLE. See Other usage constraints for Address breakpoints.

A single Context matching breakpoint with linking enabled might be linked to by all, or any combination of, the following:

- Multiple Linked Address breakpoints.
- Multiple Linked watchpoints.

If no linked breakpoints and no linked watchpoints link to a breakpoint programmed with linking enabled, then the breakpoint does not generate any Breakpoint exceptions or debug events.

Figure D2-1 shows an example of permitted breakpoint and watchpoint linking.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

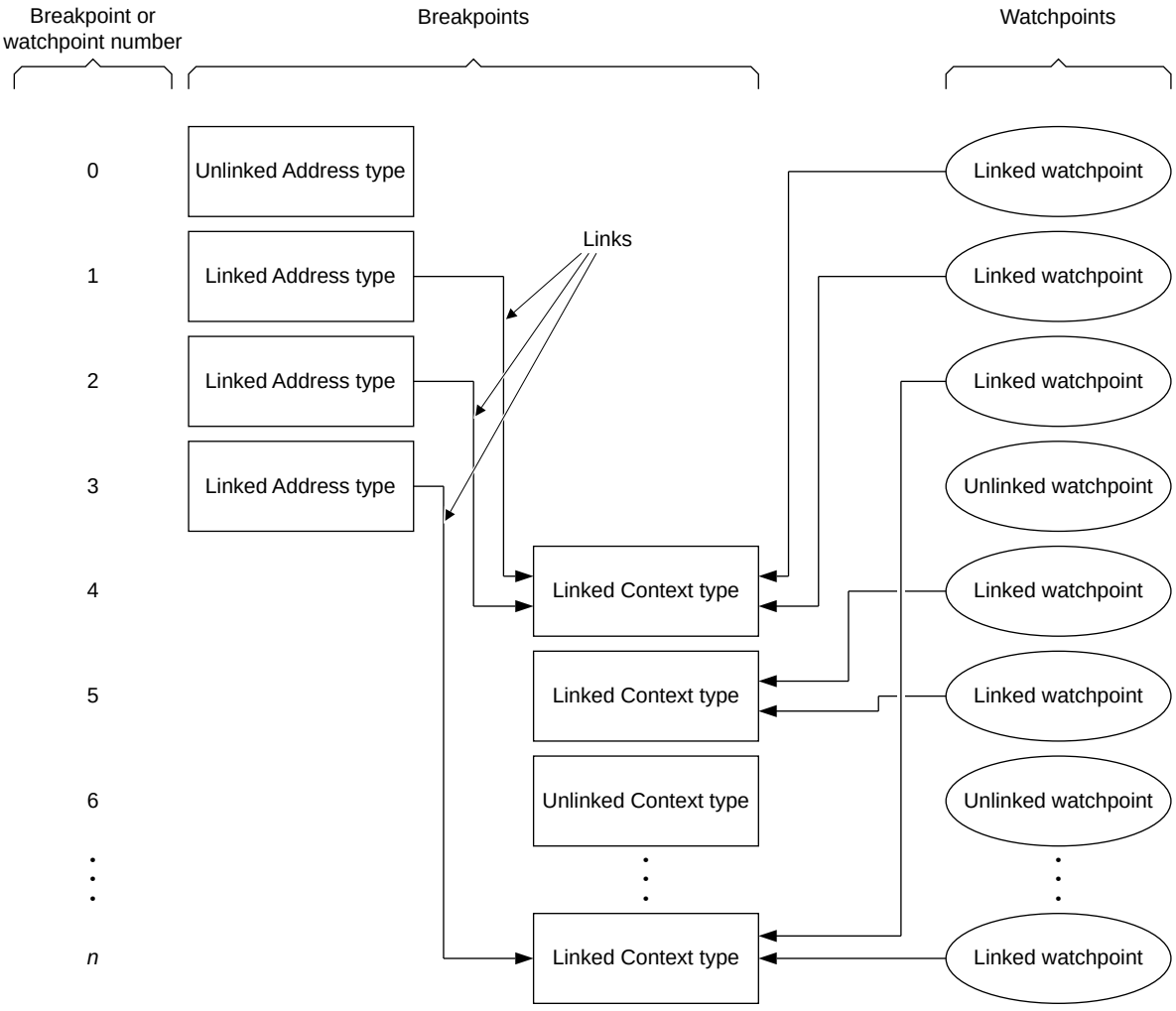


Figure D2-1 The role of linking in Breakpoint and Watchpoint exception generation

In Figure D2-1, each Linked Address breakpoint can generate a Breakpoint exception only if the comparisons made by both it, and the breakpoint that it links to, are successful. Similarly, each Linked watchpoint can only generate a Watchpoint exception if the comparisons made by both it, and the breakpoint that it links to, are successful.

D2.8.2.2 Breakpoint types defined by DBGBCR<n>_EL1.BT

The following lists provides more detail about each breakpoint type:

`0b0000`**, Unlinked Address Match breakpoint**

Generation of a Breakpoint exception depends on both:

- DBGBCR<n>_EL1.{SSC, HMC, PMC}. These define the execution conditions for which the breakpoint generates Breakpoint exceptions. See Execution conditions for which a breakpoint generates Breakpoint exceptions.
- A successful address match, as described in Breakpoint instruction address comparisons.

DBGBCR<n>_EL1.LBN for this breakpoint are ignored.

`0b0001`**, Linked Address Match breakpoint**

Generation of a Breakpoint exception depends on all of the following:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- DBGBCR<n>_EL1.{SSC, HMC, PMC} for this breakpoint. These define the execution conditions that the breakpoint generates Breakpoint exceptions for. See Execution conditions for which a breakpoint generates Breakpoint exceptions.
- A successful address match defined by this breakpoint, as described in Breakpoint instruction address comparisons.
- A successful context match defined by the Context breakpoint with linking enabled that this breakpoint links to.

DBGBCR<n>_EL1.LBN for this breakpoint selects the Context breakpoint with linking enabled that this breakpoint links to.

`0b0010`**, Context ID Match breakpoint**

BT == `0b0010` is a reserved value if the breakpoint is not a context-aware breakpoint.

For context-aware breakpoints, generation of a Breakpoint exception depends on both:

- DBGBCR<n>_EL1.{SSC, HMC, PMC}. These define the execution conditions for which the breakpoint generates Breakpoint exceptions. See Execution conditions for which a breakpoint generates Breakpoint exceptions.
- A successful Context ID match, as described in Breakpoint context comparisons.

The value of DBGBVR<n>_EL1.ContextID is compared with the current Context ID.

CONTEXTIDR_EL2 holds the current Context ID when all of:

- The implementation includes FEAT_VHE.
- The PE is executing at EL0 and HCR_EL2.TGE is 1, or the PE is executing at EL2.

Otherwise, CONTEXTIDR_EL1 holds the current Context ID.

DBGBCR<n>_EL1.{LBN, BAS} for this breakpoint are ignored

`0b0011`**, Context ID Match breakpoint with linking enabled**

BT == `0b0011` is a reserved value if the breakpoint is not a context-aware breakpoint.

For context-aware breakpoints, one of the following applies:

- If no Linked breakpoints or Linked watchpoints link to this breakpoint, then the breakpoint does not generate any debug exceptions.
- Generation of a Breakpoint exception depends on both:
  - A successful instruction address match, defined by a Linked Address breakpoint that links to this breakpoint, see Breakpoint instruction address comparisons.
  - A successful Context ID match defined by this breakpoint, as described in Breakpoint context comparisons.
- Generation of a Watchpoint exception depends on both:
  - A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see Watchpoint data address comparisons.
  - A successful Context ID match defined by this breakpoint, as described in Breakpoint context comparisons.

The value of DBGBVR<n>_EL1.ContextID is compared with the current Context ID.

CONTEXTIDR_EL2 holds the current Context ID when all of:

- The implementation includes FEAT_VHE.
- The PE is executing at EL0 and HCR_EL2.TGE is 1, or the PE is executing at EL2.

Otherwise, CONTEXTIDR_EL1 holds the current Context ID.

DBGBCR<n>_EL1.{LBN, SSC, HMC, BAS, PMC} for this breakpoint are ignored.

`0b0110`**, CONTEXTIDR_EL1 Match breakpoint**

BT == `0b0110` is a reserved value if either:

- The breakpoint is not a context-aware breakpoint.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The implementation does not include FEAT_VHE.

In an implementation that includes FEAT_VHE, for context-aware breakpoints, generation of a Breakpoint exception depends on both:

- DBGBCR<n>_EL1.{SSC, HMC, PMC}. These define the execution conditions for which the breakpoint generates Breakpoint exceptions.
- A successful Context ID match defined by this breakpoint, as described in Breakpoint context comparisons.

The Context ID check is made against the value in CONTEXTIDR_EL1. The value of DBGBVR<n>_EL1.ContextID is compared with the Context ID value held in CONTEXTIDR_EL1.

DBGBCR<n>_EL1.{LBN, BAS} for this breakpoint are ignored.

`0b0111`**, CONTEXTIDR_EL1 Match breakpoint with linking enabled**

BT == `0b0111` is a reserved value if either:

- The breakpoint is not a context-aware breakpoint.
- The implementation does not include FEAT_VHE.

In an implementation that includes FEAT_VHE, for context-aware breakpoints, one of the following applies:

- If no Linked breakpoints or Linked watchpoints link to this breakpoint, then the breakpoint does not generate any debug exceptions.
- Generation of a Breakpoint exception depends on both:
  - A successful instruction address match, defined by a Linked Address match breakpoint that links to this breakpoint, see Breakpoint instruction address comparisons.
  - A successful Context ID match defined by this breakpoint, as described in Breakpoint context comparisons.
- Generation of a Watchpoint exception depends on both:
  - A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see Watchpoint data address comparisons.
  - A successful Context ID match defined by this breakpoint, as described in Breakpoint context comparisons.

The Context ID check is made against the value in CONTEXTIDR_EL1. The value of DBGBVR<n>_EL1.ContextID is compared with the Context ID value held in CONTEXTIDR_EL1.

DBGBCR<n>_EL1.{LBNX, LBN, SSCE, SSC, HMC, BAS, PMC} for this breakpoint are ignored.

`0b1000`**, VMID Match breakpoint**

BT == `0b1000` is a reserved value if the breakpoint is not a context-aware breakpoint.

For context-aware breakpoints, generation of a Breakpoint exception depends on both:

- DBGBCR<n>_EL1.{SSC, HMC, PMC}. These define the execution conditions for which the breakpoint generates Breakpoint exceptions. See Execution conditions for which a breakpoint generates Breakpoint exceptions.
- A successful VMID match, as described in Breakpoint context comparisons.

DBGBCR<n>_EL1.{LBN, BAS} for this breakpoint are ignored.

`0b1001`**, VMID Match breakpoint with linking enabled**

BT == `0b1001` is a reserved value if the breakpoint is not a context-matching breakpoint.

For context-aware breakpoints, one of the following applies:

- If no Linked breakpoints or Linked watchpoints link to this breakpoint, then the breakpoint does not generate any debug exceptions.
- Generation of a Breakpoint exception depends on both:
  - A successful instruction address match, defined by a Linked Address Match breakpoint that links to this breakpoint. See Breakpoint instruction address comparisons.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— A successful VMID match defined by this breakpoint, as described in Breakpoint context comparisons.

- Generation of a Watchpoint exception depends on both:
  — A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see Watchpoint data address comparisons.
  — A successful VMID match defined by this breakpoint, as described in Breakpoint context comparisons.

DBGBCR<n>_EL1.{LBN, SSC, HMC, BAS, PMC} for this breakpoint are ignored.

`0b1010`**, Context ID and VMID Match breakpoint**

BT == `0b1010` is a reserved value if the breakpoint is not a context-aware breakpoint.

When EL2 is implemented, for context-aware breakpoints, generation of a Breakpoint exception depends on all of the following:

- DBGBCR<n>_EL1.{SSC, HMC, PMC}. These define the execution conditions that the breakpoint generates a Breakpoint exception for. See Execution conditions for which a breakpoint generates Breakpoint exceptions.
- A successful Context ID match, as described in Breakpoint context comparisons.
- A successful VMID match.

The value of DBGBVR<n>_EL1.ContextID is compared with CONTEXTIDR_EL1.

Breakpoint context comparisons describes the requirements for a successful Context ID match and a successful VMID match.

DBGBCR<n>_EL1.{LBN, BAS} for this breakpoint are ignored.

`0b1011`**, Context ID and VMID Match breakpoint with linking enabled**

BT == `0b1011` is a reserved value if the breakpoint is not a context-aware breakpoint.

When EL2 is implemented, for context-aware breakpoints, one of the following applies:

- If no Linked breakpoints or Linked watchpoints link to this breakpoint, then the breakpoint does not generate any debug exceptions.
- Generation of a Breakpoint exception depends on all of the following:
  — A successful instruction address match, defined by a Linked Address breakpoint that links to this breakpoint, see Breakpoint instruction address comparisons.
  — A successful Context ID match defined by this breakpoint, as described in Breakpoint context comparisons.
  — A successful VMID match defined by this breakpoint.
- Generation of a Watchpoint exception depends on all of the following:
  — A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see Watchpoint data address comparisons.
  — A successful Context ID match defined by this breakpoint, as described in Breakpoint context comparisons.
  — A successful VMID match defined by this breakpoint.

The value of DBGBVR<n>_EL1.ContextID is compared with CONTEXTIDR_EL1.

Breakpoint context comparisons describes the requirements for a successful Context ID match and a successful VMID match by this breakpoint.

DBGBCR<n>_EL1.{LBN, SSC, HMC, BAS, PMC} for this breakpoint are ignored.

`0b1100`**, CONTEXTIDR_EL2 Match breakpoint**

BT == `0b1100` is a reserved value if either:

- The breakpoint is not a context-aware breakpoint.
- FEAT_VHE is not implemented and FEAT_Debugv8p2 is not implemented, which means the implementation does not include CONTEXTIDR_EL2.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

In an implementation in which FEAT_VHE is implemented or FEAT_Debugv8p2 is implemented, for context-aware breakpoints, generation of a Breakpoint exception depends on both:

- DBGBCR<n>_EL1.{SSC, HMC, PMC}. These define the execution conditions for which the breakpoint generates Breakpoint exceptions.
- A successful CONTEXTIDR_EL2 match, as described in Breakpoint context comparisons.

The Context ID check is made against the value in CONTEXTIDR_EL2. The value of DBGBVR<n>_EL1 is compared with the Context ID value held in CONTEXTIDR_EL2.

DBGBCR<n>_EL1.{LBN, BAS} for this breakpoint are ignored.

`0b1101`**, CONTEXTIDR_EL2 Match breakpoint with linking enabled**

BT == `0b1101` is a reserved value if either:

- The breakpoint is not a context-aware breakpoint.
- FEAT_VHE is not implemented and FEAT_Debugv8p2 is not implemented, which means the implementation does not include CONTEXTIDR_EL2.

In an implementation in which FEAT_VHE is implemented or FEAT_Debugv8p2 is implemented, for context-aware breakpoints, either:

- If no Linked breakpoints or Linked watchpoints link to this breakpoint, then the breakpoint does not generate any debug exceptions.
- Generation of a Breakpoint exception depends on both:
  - A successful instruction address match, defined by a Linked Address match breakpoint that links to this breakpoint, see Breakpoint instruction address comparisons.
  - A successful CONTEXTIDR_EL2 match, as described in Breakpoint context comparisons.
- Generation of a Watchpoint exception depends on both:
  - A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see Watchpoint data address comparisons.
  - A successful CONTEXTIDR_EL2 match, as described in Breakpoint context comparisons.

The Context ID check is made against the value in CONTEXTIDR_EL2. The value of DBGBVR<n>_EL1 is compared with the Context ID value held in CONTEXTIDR_EL2.

DBGBCR<n>_EL1.{LBN, SSC, HMC, BAS, PMC} for this breakpoint are ignored.

`0b1110`**, Full Context ID Match breakpoint**

BT == `0b1110` is a reserved value if either:

- The breakpoint is not a context-aware breakpoint.
- FEAT_VHE is not implemented and FEAT_Debugv8p2 is not implemented, which means the implementation does not include CONTEXTIDR_EL2.

In an implementation in which FEAT_VHE is implemented or FEAT_Debugv8p2 is implemented, for context-aware breakpoints, generation of a Breakpoint exception depends on both:

- DBGBCR<n>_EL1.{SSC, HMC, PMC}. These define the execution conditions for which the breakpoint generates Breakpoint exceptions.
- A successful Context ID match, as described in Breakpoint context comparisons.

The Context ID check is made against the values in both CONTEXTIDR_EL1 and CONTEXTIDR_EL2. The value of DBGBVR<n>_EL1[31:0] is compared with the Context ID value held in CONTEXTIDR_EL1, and the value of DBGBVR<n>_EL1[63:32] is compared with the Context ID value held in CONTEXTIDR_EL2. Both comparisons must match for the Context ID check.

DBGBCR<n>_EL1.{LBN, BAS} for this breakpoint are ignored.

`0b1111`**, Full Context ID Match breakpoint with linking enabled**

BT == `0b1111` is a reserved value if either:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The breakpoint is not a context-aware breakpoint.
- FEAT_VHE is not implemented and FEAT_Debugv8p2 is not implemented, which means the implementation does not include CONTEXTIDR_EL2.

In an implementation in which FEAT_VHE is implemented or FEAT_Debugv8p2 is implemented, for context-aware breakpoints, one of the following applies:

- If no Linked breakpoints or Linked watchpoints link to this breakpoint, then the breakpoint does not generate any debug exceptions.
- Generation of a Breakpoint exception depends on both:
  - A successful instruction address match, defined by a Linked Address match breakpoint that links to this breakpoint, see Breakpoint instruction address comparisons.
  - A successful Context ID match, as described in Breakpoint context comparisons.
- Generation of a Watchpoint exception depends on both:
  - A successful data address match, defined by a Linked watchpoint that links to this breakpoint, see Watchpoint data address comparisons.
  - A successful Context ID match, as described in Breakpoint context comparisons.

The Context ID check is made against the values in both CONTEXTIDR_EL1 and CONTEXTIDR_EL2. The value of DBGBVR<n>_EL1[31:0] is compared with the Context ID value held in CONTEXTIDR_EL1, and the value of DBGBVR<n>_EL1[63:32] is compared with the Context ID value held in CONTEXTIDR_EL2. Both comparisons must match for the Context ID check.

DBGBCR<n>_EL1.{LBN, SSC, HMC, BAS, PMC} for this breakpoint are ignored.

D2.8.3 Execution conditions for which a breakpoint generates Breakpoint exceptions

Each breakpoint can be programmed so that it generates Breakpoint exceptions only for certain execution conditions. For example, a breakpoint might be programmed to generate Breakpoint exceptions only when the PE is executing at EL0 in Secure state.

In the table:

Y or - Means that a breakpoint programmed with the values of HMC, SSCE, SSC, and PMC shown in that row:

Y Can generate Breakpoint exceptions in that Exception level and Security state.

- Cannot generate Breakpoint exceptions in that Exception level and Security state.

Table D2-5 Summary of breakpoint HMC, SSC, and PMC encodings

DBGBCR<n>_EL1			Exception Level		
HMC	SSC	PMC	EL2	EL1	EL0
0	00	01	-	Y	-
0	00	10	-	-	Y
0	00	11	-	Y	Y
0	10	01	-	Y	-
0	10	10	-	-	Y
0	10	11	-	Y	Y
0	11	00	Y	-	-
0	11	01	Y	Y	-

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DBGBCR<n>_EL1			Exception Level		
HMC	SSC	PMC	EL2	EL1	EL0
0	11	11	Y	Y	Y
1	00	01	Y	Y	-
1	00	11	Y	Y	Y
1	01	00	Y	-	-
1	01	01	Y	Y	-
1	01	11	Y	Y	Y
1	10	01	Y	Y	-
1	10	11	Y	Y	Y
1	11	00	Y	-	-
1	11	01	Y	Y	-
1	11	11	Y	Y	Y
1	01	00	Y	-	-

All combinations of HMC, SSC, and PMC that Table D2-5 does not show are reserved. A combination in Table D2-5 might be reserved if an Exception level or Security state is not implemented. For information about which combinations of HMC, SSCE, SSC and PMC are reserved if an Exception level or Security state are not implemented, See Reserved DBGBCR<n>_EL1.{SSC, HMC, PMC} values.

D2.8.4 Breakpoint instruction address comparisons

In this subsection, the term `AddrTop` represents the most significant bit of a virtual address used by breakpoint data address comparisons. `AddrTop` is:

- 55, if address tagging is used for the address. See Address tagging.
- 63, otherwise.

Note

When stage 1 translation is enabled, in AArch64 state, the virtual address size is determined by the configured input address size for the stage 1 translation. Software can configure a smaller address width for a virtual address, see Input address size configuration. Attempting to translate an address that is larger than the configured input address size generates a Translation fault.
When stage 1 translation is disabled, using an address that is larger than the implemented PA size generates an Address size fault. The implemented PA size is IMPLEMENTATION DEFINED, as described in Implemented physical address size. These faults have a higher priority than breakpoints.

An address comparison is successful if bits [`AddrTop`:2] of the current instruction virtual address are equal to DBGBVR<n>_EL1[`AddrTop`:2].

Note

DBGBVR<n>_EL1 is a 64-bit register. The most significant bits of this register are sign-extension bits. DBGBVR<n>_EL1[1:0] are RES0 and are ignored.

All instructions are word-aligned and DBGBCR<n>_EL1.BAS is RES1.

If the implementation is an AArch64-only implementation, all instructions are word-aligned and DBGBCR<n>_EL1.BAS is RES1.

Figure D2-2 shows a summary of when Address Match breakpoints programmed with particular BAS values generate Breakpoint exceptions. The figure contains four parts:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- A column showing the row number, on the left.
- An instruction set and instruction size table.
- A location of instruction figure.
- A BAS field values table, on the right.

To use the figure, read across the rows. For example, row 7 shows that a breakpoint with DBGBCR<n>_EL1.BAS programmed as either `0b0011` or `0b1111` generates Breakpoint exceptions for A64 instructions. A64 instructions are always at word-aligned addresses.

Note

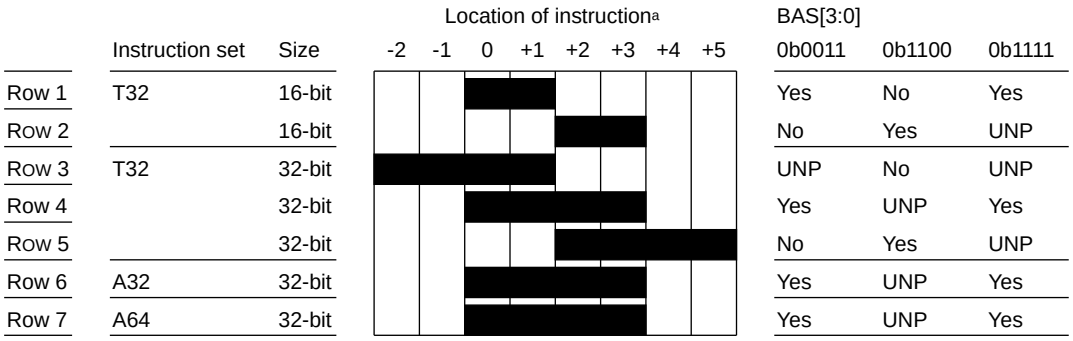
To breakpoint on an A64 instruction, Arm recommends that the debugger programs DBGBCR<n>_EL1.BAS as `0b1111`.

In the figure:

Yes Means that the breakpoint generates a Breakpoint exception.

No Means that the breakpoint does not generate a Breakpoint exception.

UNP Means that is it CONSTRAINED UNPREDICTABLE whether the breakpoint generates a Breakpoint exception. See Other usage constraints for Address breakpoints.



a. 0 means the word-aligned address held in the DBGBVR<n>_EL1[maxAddressSize:2]:00.
The other locations are as follows:
- -2 means ((DBGBVR<n>_EL1[maxAddressSize:2]:00) - 2).
- -1 means ((DBGBVR<n>_EL1[maxAddressSize:2]:00) - 1).
- ...
- ...
- +5 means ((DBGBVR<n>_EL1[maxAddressSize:2]:00) + 5).

The solid areas show the location of the instruction.

Figure D2-2 Summary of BAS field meanings for Address Match breakpoints

D2.8.5 Breakpoint context comparisons

The breakpoint type defined by DBGBCR<n>_EL1.BT determines what context comparison is required, if any. ????? Table D2-6 shows the BT values that require a comparison, and the match required for the comparison to be successful.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D2-6 Breakpoint Context ID and VMID comparison tests

DBGBVR<n>_EL1.BT	Test required for successful context comparison
0b001x	• When FEAT_VHE is implemented, EL2 is using AArch64, the Effective value of HCR_EL2.E2H is 1, and either the PE is executing at EL0 with HCR_EL2.TGE set to 1, or the PE is executing at EL2, CONTEXTIDR_EL2 must match the DBGBVR<n>_EL1.ContextID value. • Otherwise, CONTEXTIDR_EL1 must match the DBGBVR<n>_EL1.ContextID value.
0b011x	CONTEXTIDR_EL1 must match the DBGBVR<n>_EL1.ContextID value.
0b100x	VSCTLR_EL2.VMID must match the DBGBVR<n>_EL1.VMID value.
0b101x	CONTEXTIDR_EL1 must match the DBGBVR<n>_EL1.ContextID value and VSCTLR_EL2.VMID must match the DBGBVR<n>_EL1.VMID value.
0b110x	CONTEXTIDR_EL2 must match the DBGBVR<n>_EL1.ContextID2 value, DBGBVR<n>_EL1[63:32].
0b111x	Both: • CONTEXTIDR_EL1 must match the DBGBVR<n>_EL1.ContextID value, DBGBVR<n>_EL1[31:0]. • CONTEXTIDR_EL2 must match the DBGBVR<n>_EL1.ContextID2 value, DBGBVR<n>_EL1[63:32].

No Context ID or VMID comparison is required for other valid DBGBCR<n>_EL1, Debug Breakpoint Control Registers, n = 0 - 63.BT values.

Context breakpoints do not generate Breakpoint exceptions when any of:

- The comparison uses the value of CONTEXTIDR_EL1 when the PE is executing at EL1 or EL0.
- The comparison uses the value of CONTEXTIDR_EL2 when the PE is executing at EL2.
- The comparison uses the current VMID value and the PE is executing at EL2.

Note

- For Context matching breakpoints with linking enabled, DBGBCR<n>_EL1.{LBN, SSC, HMC, PMC} are RES0 and are ignored.

D2.8.6 Breakpoint usage constraints

See the following sections:

- Reserved DBGBCR<n>_EL1.BT values.
- Reserved DBGBCR<n>_EL1.{SSC, HMC, PMC} values.
- Usage constraints on DBGBCR<n>_EL1.LBN values.
- Other usage constraints for Address breakpoints.
- Other usage constraints for Context breakpoints.

D2.8.6.1 Reserved DBGBCR<n>_EL1.BT values

Table D2-7 shows when particular DBGBCR<n>_EL1.BT values are reserved.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D2-7 Reserved BT values

BT value	Breakpoint type	Reserved
0b001x	Context ID Match	If the breakpoint is not *context-aware*.
0b011x	CONTEXTIDR_EL1 Match	If the breakpoint is not *context-aware*.
0b110x	CONTEXTIDR_EL2 Match	If the breakpoint is not *context-aware*.
0b111x	Full Context ID Match	

If a breakpoint is programmed with one of these reserved BT values:

- The breakpoint must behave as if it is either:
  - Disabled.
  - Programmed with a BT value that is not reserved, other than for a direct or external read of DBGBCR<n>_EL1.
- For a direct or external read of DBGBCR<n>_EL1, if the reserved {BT2, BT} value:
  - Has no function for any execution conditions, the value read back is UNKNOWN.
  - Has a function for execution conditions other than the current execution conditions, the value read back is the value written. This permits software to save and restore the {BT2, BT} value so that the breakpoint functions for the other execution conditions.

The behavior of breakpoints with reserved {BT2, BT} values might change in future revisions of the architecture. For this reason, software must not rely on the behavior described here.

D2.8.6.2 Reserved DBGBCR<n>_EL1.{SSC, HMC, PMC} values

Table D2-8 shows when particular combinations of DBGBCR<n>_EL1.{SSCE, SSC, HMC, PMC} are reserved in stage 1 of an AArch64 translation regime.

Table D2-8 Reserved HMC, SSCE, SSC, and PMC combinations

HMC, SSC, and PMC combination	Reserved
All combinations with and SSC set to 0b01 or 0b10, except for the combination with HMC set to 1, SSC set to 0b01, and PMC set to 0b00.	
Combinations not included in Table D2-5.	Always.

For all breakpoints except Context matching breakpoints with linking enabled, if a breakpoint is programmed with one of these reserved combinations:

- If the reserved combination has a function for other execution conditions:
  - The breakpoint must behave as if it is disabled.
  - A direct or external read of DBGBCR<n>_EL1.{SSC, HMC, PMC} returns the values written. This means that software can save and restore the combination so that the breakpoint can function for the other execution conditions.
- If the reserved combination does not have a function for other execution conditions:
  - It must behave either as if it is programmed with a combination that is not reserved or as if it is disabled.
  - A direct or external read of DBGBCR<n>_EL1.{SSC, HMC, PMC} returns UNKNOWN values.

If the breakpoint is a Context matching breakpoint with linking enabled, then:

- The values of HMC, SSC, and PMC are ignored.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- A direct or external read of DBGBCR<n>_EL1.{SSC, HMC, PMC} returns UNKNOWN values

The behavior of breakpoints with reserved combinations of HMC, SSC, and PMC might change in future revisions of the architecture. For this reason, software must not rely on the behavior described here.

D2.8.6.3 Usage constraints on DBGBCR<n>_EL1.LBN values

For Linked Address breakpoints

A Linked Address breakpoint must link to a *context-aware* breakpoint. For a Linked Address breakpoint, any DBGBCR<n>_EL1.LBN value that is not for a *context-aware* breakpoint is reserved.

If a Linked Address breakpoint links to a breakpoint that is not implemented, or that is not *context-aware*, then reads of DBGBCR<n>_EL1.LBN return an UNKNOWN value and behavior is CONSTRAINED UNPREDICTABLE. The Linked Address breakpoint behaves as if it is either:

- Disabled.
- Linked to an UNKNOWN *context-aware* breakpoint.

If a Linked Address breakpoint links to a breakpoint that is implemented and that is *context-aware*, but that is either not enabled or not programmed as a Context matching breakpoint with linking enabled, it behaves as if it is disabled.

For all other breakpoint types

DBGBCR<n>_EL1.LBN reads UNKNOWN and its value is ignored.

D2.8.6.4 Other usage constraints for Address breakpoints

For all Address breakpoints

- DBGBVR<n>_EL1[1:0] are RES0 and are ignored.

D2.8.6.5 Other usage constraints for Context breakpoints

For all Context breakpoints

Any bits of DBGBVR<n>_EL1 that are not used to specify Context ID or VMID are RES0 and are ignored.

For breakpoints with linking enabled

If no Linked Address breakpoints or Linked watchpoints link to a breakpoint with linking enabled, the breakpoint with linking enabled does not generate any Breakpoint exceptions.

D2.8.7 Preferred return address

The preferred return address of a Breakpoint exception is the address of the instruction that was not executed because the PE took the Breakpoint exception instead.

This means that the preferred return address is the address of the instruction that caused the exception.

D2.8.8 Pseudocode description of Breakpoint exceptions taken from an AArch64 stage 1 translation regime

`AArch64.BreakpointValueMatch()` tests the value in DBGBVR<n>_EL1.

`AArch64.StateMatch`() tests the values in DBGBCR<n>_EL1.{ SSC, HMC, PMC} and, if the breakpoint links to a breakpoint with linking enabled, also tests the Context matching breakpoint.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`AArch64.BreakpointMatch()` tests a committed instruction against all breakpoints.

`AArch64.CheckBreakpoint()` generates a Breakpoint exception if all of the following are true:

- MDSCR_EL1.MDE is 1.
- Debug exceptions are enabled from the current Exception level and Security state. See Enabling debug exceptions from the current Exception level and Security state.
- All of the conditions required for Breakpoint exception generation are met. See About Breakpoint exceptions.

Note

`AArch64.CheckBreakpoint()` might halt the PE and cause it to enter Debug state. External debug uses Debug state.

`AArch64.BreakpointException()` is called to generate a Breakpoint exception.

These functions are defined in R-profile Architecture Pseudocode.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D2.9 Watchpoint exceptions

This section describes Watchpoint exceptions in stage 1 of an AArch64 translation regime.

This section contains the following subsections:

- About Watchpoint exceptions.
- Watchpoint types and linking of watchpoints.
- Execution conditions for which a watchpoint generates Watchpoint exceptions.
- Watchpoint data address comparisons.
- Watchpoint behavior for certain instruction classes.
- Watchpoint usage constraints.
- Exception syndrome information, fault address information, and preferred return address.
- Pseudocode description of Watchpoint exceptions taken from AArch64 state.

D2.9.1 About Watchpoint exceptions

A *watchpoint* is an event that results from the execution of an instruction, based on a data address. Watchpoints are also known as *data breakpoints*.

A watchpoint operates as follows:

1. A debugger programs the watchpoint with a data address, or a data address range.
2. The watchpoint generates a *Watchpoint debug event* on an access to the address, or any address in the address range.

A watchpoint never generates a Watchpoint debug event on an instruction fetch.

An implementation can include between 2-16 watchpoints. In an implementation, ID_AA64DFR0_EL1.WRPs and EDDFR1.WRPs show how many are implemented.

To use an implemented watchpoint, a debugger programs the following registers for the watchpoint:

- The *Watchpoint Control Register*, DBGWCR<n>_EL1. This contains controls for the watchpoint, for example an enable control.
- The *Watchpoint Value Register*, DBGWVR<n>_EL1. This holds the data virtual address used for watchpoint matching.

These registers are numbered, so that:

- DBGWCR0_EL1 and DBGWVR0_EL1 are for watchpoint number zero.
- DBGWCR1_EL1 and DBGWVR1_EL1 are for watchpoint number one.
- . . .
- DBGWCR<n-1>_EL1 and DBGWVR<n-1>_EL1 are for watchpoint number (n-1).

A watchpoint can:

- Be programmed to generate Watchpoint debug events on read accesses only, on write accesses only, or on both types of access.
- Link to a *Context matching breakpoint with linking enabled*, so that a Watchpoint debug event is generated only if the PE is in a particular context when the address match occurs.
- Be programmed as an address match watchpoint or an address mismatch watchpoint. See Watchpoint data address comparisons.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A single watchpoint can be programmed to match or mismatch on one or more address bytes. A watchpoint generates a Watchpoint debug event on an access to any byte that it is watching. The number of bytes a watchpoint is watching is either:

- One to eight bytes, provided that these bytes are contiguous and that they are all in the same naturally-aligned doubleword. A debugger uses the *Byte Address Select* field, DBGWCR<n>_EL1.BAS, to select the bytes. See Programming a watchpoint range with eight bytes or fewer.
- Eight bytes to 2GB, provided that both of the following are true:
  - The number of bytes is a power-of-two.
  - The range starts at an address that is aligned to the range size.

  A debugger uses the *MASK* field, DBGWCR<n>_EL1.MASK, to program a watchpoint with eight bytes to 2GB. See Programming a watchpoint range with eight or more bytes.

A debugger must use either the BAS field or the MASK field. If it uses both, whether the watchpoint generates Watchpoint debug events is CONSTRAINED UNPREDICTABLE. See Programming dependencies of the BAS and MASK fields.

For each memory access, all of the watchpoints are tested. When a watchpoint is tested, it generates a Watchpoint debug event if all of the following are true:

- The conditions specified in the DBGWCR<n>_EL1 are met. This includes all of the following:
  - The watchpoint is enabled. See DBGWCR<n>_EL1.E.
  - The type of the access. That is, whether the access is a load, a store, or both. See DBGWCR<n>_EL1.LST.
  - The Security state, Exception level, and privilege of the access. See Execution conditions for which a watchpoint generates Watchpoint exceptions.
  - If the watchpoint links to a breakpoint with linking enabled, the comparison or comparisons made by the breakpoint. See Rules for linking watchpoints.

  Together, these define the *context* of the data access that the watchpoint matches.
- The comparison with the address held in the DBGWVR<n>_EL1 is successful. See Watchpoint data address comparisons.
- The instruction that initiates the memory access is committed for execution.
- The instruction that initiates the memory access passes its Condition code check.
- If the access is due to a System register access instruction executed at EL1 and transformed into a memory access by the mechanism described in Enhanced support for nested virtualization and one of the following is true:
  - EDSCR.HDE is set to 1 and halting is allowed.
  - Debug exceptions are enabled at EL2. For more information, see Enabling debug exceptions from the current Exception level and Security state.

If multiple watchpoints can generate a Watchpoint debug event for an access, then, for all watchpoints that are enabled and match the context of the data access:

- The Boolean equality comparison results of all the address match watchpoints are OR'd together to generate a first result.
- The Boolean inequality comparison results of all the address mismatch watchpoints are AND'd together to generate a second result.

A watchpoint debug event is generated if any of the following apply:

- The first result is true and all enabled watchpoints are address match watchpoints.
- The second result is true and all enabled watchpoints are address mismatch watchpoints.
- Both the first and second results are true, and both address match watchpoints and address mismatch watchpoints are enabled.

The pseudocode function `AArch64.CheckWatchpoint()` describes the details for combining the results of individual watchpoints to determine the final result of watchpoint match.

If halting is allowed and EDSCR.HDE is 1, Watchpoint debug events cause entry to Debug state. Otherwise, if debug exceptions are:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Enabled, Watchpoint debug events generate Watchpoint exceptions.
- Disabled, Watchpoint debug events are ignored.

Note

The remainder of this Watchpoint Exceptions section, including all subsections, describes watchpoints as generating Watchpoint exceptions.
However, unless specified otherwise, the behavior described also applies if watchpoints are causing entry to Debug state.

The debug exception enable controls describes the enable controls for Watchpoint debug events.

D2.9.2 Watchpoint types and linking of watchpoints

When a debugger programs a watchpoint, it must program that watchpoint so that:

- It is either:
  - Used in isolation. In this case, the watchpoint is called an *Unlinked watchpoint*.
  - Linked to a breakpoint with linking enabled. In this case, the watchpoint is called a *Linked watchpoint*.
- And is either:
  - An address match watchpoint.
  - An address mismatch watchpoint.

When a Linked watchpoint links to an Address matching breakpoint with linking enabled, the Linked watchpoint generates a Watchpoint exception only if the address of the instruction that made the matching access was executed from a specific address or in range of addresses.

When a Linked watchpoint links to a Context matching breakpoint with linking enabled, or a Linked watchpoint links to a Linked Address matching breakpoint with linking enabled that links to a Context matching breakpoint with linking enabled, the Linked watchpoint generates a Watchpoint exception only if the PE is in a particular context when the data address match occurs.

Example D2-2 Linking a watchpoint and an Address matching breakpoint

For example, a debugger might:

1. Program watchpoint number one with a data address.
2. Program breakpoint number five to be a *Linked VMID Match breakpoint*.
3. Link the watchpoint and the breakpoint together. A Watchpoint exception is generated only if both the data address matches and the VMID matches.

D2.9.2.1 Rules for linking watchpoints

The rules for watchpoint linking are as follows:

- Only Linked watchpoints can be linked.
- A Linked watchpoint can be linked to any breakpoint with linking enabled.
- A Linked watchpoint can link to any type of breakpoint with linking enabled. The *Linked Breakpoint Number* field, DBGWCR<n>_EL1.{ LBN}, for the Linked watchpoint specifies the particular breakpoint that the Linked watchpoint links to, and:
  - DBGWCR<n>_EL1.{SSC, HMC, PAC} for the Linked watchpoint defines the execution conditions that the watchpoint generates Watchpoint exceptions for. See Execution conditions for which a watchpoint generates Watchpoint exceptions.
  - DBGBCR<n>_EL1.{ SSC, HMC, PMC} for the breakpoint with linking enabled are ignored.
- A Linked watchpoint cannot link to another watchpoint. The { LBN} fields always specify a breakpoint.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If any of the following apply, then the behavior of the Linked watchpoint is CONSTRAINED UNPREDICTABLE:
  - The Linked watchpoint is linked to a breakpoint that does not support linking.
  - The Linked watchpoint is linked to a breakpoint that is not context aware.
  - The Linked watchpoint is linked to an Address matching breakpoint.
  - The Linked watchpoint is linked to an unimplemented breakpoint.
  - The Linked watchpoint or Linked breakpoint is linked to a breakpoint that does not have linking enabled.

  See Watchpoint usage constraints.

- Multiple Linked watchpoints can link to a single breakpoint with linking enabled.

  Note

  Multiple Address breakpoints can also link to a single breakpoint with linking enabled. Breakpoint exceptions describes breakpoints.

- If an instruction address generates a breakpoint match for more than one enabled Address matching breakpoint with linking enabled, then it is CONSTRAINED UNPREDICTABLE whether Watchpoint exceptions and debug events are generated.

Figure D2-1 shows an example of permitted watchpoint linking.

D2.9.3 Execution conditions for which a watchpoint generates Watchpoint exceptions

Each watchpoint can be programmed so that it generates Watchpoint exceptions only for certain execution conditions. For example, a watchpoint might be programmed to generate Watchpoint exceptions only for Non-secure EL2 accesses.

DBGWCR<n>_EL1.{SSC, HMC, PAC} define the execution conditions a watchpoint generates Watchpoint exceptions for, as follows:

Security State Control, SSC

SSC control whether the watchpoint generates Watchpoint exceptions in only one Security state or in multiple Security states.

Note

This is determined by the Security state of the PE, not from the physical address space attribute returned by the translation of the virtual address on which the watchpoint is set.

Higher Mode Control, HMC, and Privileged Access Control, PAC

HMC and PAC together control which Exception levels and privilege levels the watchpoint generates Watchpoint exceptions in.

The PAC control relates to the privilege of the memory access, not to the Exception level at which the access was made.

Example D2-3 PAC condition for generating a Watchpoint exception

This means that, if the PE executes a Load unprivileged or Store unprivileged instruction at EL1, the resulting data access triggers a watchpoint only if both:

- PAC is programmed to a value that generates watchpoints on EL0 accesses.
- All other conditions for generating the watchpoint are met.

Example A64 Load unprivileged and Store unprivileged instructions are `LDTR` and `STTR`.

Table D2-9 shows the valid combinations of HMC, SSC, and PAC, and for each combination shows which Exception levels watchpoints generate Watchpoint exceptions in.

In the table:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Y or - Means that a watchpoint programmed with the values of HMC, SSC, and PAC shown in that row:

Y Can generate Watchpoint exceptions in that Exception level and Security state.

- Cannot generate Watchpoint exceptions in that Exception level and Security state.

Table D2-9 Summary of watchpoint HMC, SSC, and PAC encodings

HMC	SSC	PAC	EL2	EL1	EL0
0	00	01	-	Y	-
0	00	10	-	-	Y
0	00	11	-	Y	Y
0	01	01	-	Y	-
0	01	10	-	-	Y
0	01	11	-	Y	Y
0	11	00	Y	-	-
0	11	01	Y	Y	-
0	11	11	Y	Y	Y
1	00	01	Y	Y	-
1	00	11	Y	Y	Y
1	01	00	Y	-	-
1	01	01	-	Y	-
1	01	11	-	Y	Y
1	10	11	Y	-	Y
1	11	00	Y	-	-
1	11	01	Y	Y	-
1	11	11	Y	Y	Y

All combinations of HMC, SSC, and PAC that Table D2-9 does not show are reserved.

For information about which combinations of HMC, SSC and PAC are reserved if an Exception level or Security state are not implemented or enabled, see Reserved DBGWCR<n>_EL1.{SSC, HMC, PAC} values.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D2.9.4 Watchpoint data address comparisons

In this subsection, the term `AddrTop` represents the most significant bit of a virtual address used by watchpoint data address comparisons. `AddrTop` is:

- 55, if address tagging is used for the address. See Address tagging.
- 63, otherwise.

Note

When stage 1 translation is enabled, in AArch64 state, the virtual address size is determined by the configured input address size for the stage 1 translation. Software can configure a smaller address width for a virtual address. See Input address size configuration. Attempting to translate an address that is larger than the configured input address size generates a Translation fault.
When stage 1 translation is disabled, using an address that is larger than the implemented PA size generates an Address size fault. The implemented PA size is IMPLEMENTATION DEFINED, as described in Implemented physical address size. These faults have a higher priority than watchpoints.

Bits [`AddrTop`:2] of the data address are compared with DBGWVR<n>_EL1[`AddrTop`:2], taking into account all of the following:

- The size of the access. See Size of the data access.
- The bytes selected by DBGWVR<n>_EL1.BAS. See Programming a watchpoint range with eight bytes or fewer.
- Any address ranges indicated by DBGWVR<n>_EL1.MASK. See Programming a watchpoint range with eight or more bytes.

The comparison is successful if any of following are true:

- The watchpoint is an address match watchpoint, and the two values being compared match.
- The watchpoint is an address mismatch watchpoint, and the two values being compared do not match.

Note

- DBGWVR<n>_EL1 is a 64-bit register. The most significant bits of this register are sign-extension bits.
- DBGWVR<n>_EL1[1:0] are RES0 and are ignored.

D2.9.4.1 Size of the data access

Because watchpoints can be programmed to generate Watchpoint exceptions on individual bytes, the size of each data access must be taken into account. The address of every byte in a data access is compared, and:

- In the case of an address match watchpoint, the watchpoint generates a Watchpoint exception if any byte address matches. See Example D2-4.
- In the case of an address mismatch watchpoint, the watchpoint generates a Watchpoint exception if any byte address does not match. See Example D2-5.

Example D2-4

1. A debugger programs an address match watchpoint to generate Watchpoint exceptions only when the byte at address `0x1009` is accessed.
2. The PE accesses the unaligned doubleword starting at address `0x1003`.

In this scenario, the watchpoint generates a Watchpoint exception.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example D2-5

1. A debugger programs an address mismatch watchpoint to generate Watchpoint exceptions when any address other than the byte at address `0x1009` is accessed.
2. The PE accesses the unaligned doubleword starting at address `0x1003`.

In this scenario, the watchpoint generates a Watchpoint exception.

The size of data accesses initiated by `DC ZVA` instructions is the DC ZVA block size that DCZID_EL0.BS defines.

The size of data accesses initiated by `DC IVAC` instructions is an IMPLEMENTATION DEFINED size that is both:

- From the inclusive range between:
  - The size that CTR_EL0.DminLine defines.
  - 2KB.
- A power-of-two.

For both of these instructions:

- The lowest address accessed by the instruction is the address supplied to the instruction, rounded down to the nearest multiple of the access size initiated by that instruction.
- The highest address accessed is (size - 1) bytes above the lowest address accessed.

See also Watchpoint behavior on accesses by the DC IVAC instruction and the DC ZVA instruction.

D2.9.4.2 Programming a watchpoint range with eight bytes or fewer

The Byte Address Select field, DBGWCR<n>_EL1.BAS, selects bytes in the doubleword starting at the address contained in DBGWVR<n>_EL1.

If the address programmed into the DBGWVR<n>_EL1 is:

- Doubleword-aligned:
  - All eight bits of DBGWCR<n>_EL1.BAS are used.
- Word-aligned but not doubleword-aligned:
  - Only DBGWCR<n>_EL1.BAS[3:0] are used. In this case, DBGWCR<n>_EL1.BAS[7:4] are RES0.

An address match watchpoint generates a Watchpoint exception when an access accesses any byte address that DBGWCR<n>_EL1.BAS selects. An address mismatch watchpoint generates a Watchpoint exception when an access accesses any byte address that DBGWCR<n>_EL1.BAS does not select.

Table D2-10 Supported BAS values when the DBGWVR<n>_EL1 address alignment is doubleword

BAS value	Description
`0b00000000`	No byte selected.
BAS[0] == 1	Byte at address DBGWVR<n>_EL1[`AddrTop`:3]:`000` selected.
BAS[1] == 1	Byte at address DBGWVR<n>_EL1[`AddrTop`:3]:`001` selected.
BAS[2] == 1	Byte at address DBGWVR<n>_EL1[`AddrTop`:3]:`010` selected.
BAS[3] == 1	Byte at address DBGWVR<n>_EL1[`AddrTop`:3]:`011` selected.
BAS[4] == 1	Byte at address DBGWVR<n>_EL1[`AddrTop`:3]:`100` selected.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

BAS value	Description
BAS[5] == 1	Byte at address DBGWVR<n>_EL1[AddrTop:3]:101 selected.
BAS[6] == 1	Byte at address DBGWVR<n>_EL1[AddrTop:3]:110 selected.
BAS[7] == 1	Byte at address DBGWVR<n>_EL1[AddrTop:3]:111 selected.

Table D2-11 Supported BAS values when the DBGWVR<n>_EL1 address alignment is word

BAS value[a]	Description
0b00000000	No byte selected.
BAS[0] == 1	Byte at address DBGWVR<n>_EL1[AddrTop:2]:00 selected.
BAS[1] == 1	Byte at address DBGWVR<n>_EL1[AddrTop:2]:01 selected.
BAS[2] == 1	Byte at address DBGWVR<n>_EL1[AddrTop:2]:10 selected.
BAS[3] == 1	Byte at address DBGWVR<n>_EL1[AddrTop:2]:11 selected.

a. DBGWCR<n>_EL1.BAS[7:4] are RES0.

If the BAS field selects more than one byte, the bytes it selects must be contiguous. For watchpoint behavior when its BAS field selects non-contiguous bytes, see Other usage constraints.

When programming the BAS field with anything other than 0b11111111, a debugger must program DBGWCR<n>_EL1.MASK to 0b00000. See Programming dependencies of the BAS and MASK fields.

A watchpoint generates a Watchpoint exception whenever a watched byte is accessed, even if:

- The access size is smaller or larger than the address region being watched.
- The access is misaligned, and the base address of the access is not in the doubleword or word of memory addressed by the DBGWVR<n>_EL1[AddrTop:3]. See Example D2-4.

Example D2-6

The following are some example configurations of the BAS field:

- To program an address match watchpoint to generate a Watchpoint exception on the byte at address 0x1003, program:
  - DBGWVR<n>_EL1 with 0x1000.
  - DBGWCR<n>_EL1.BAS to be 0b00001000.
- To program an address match watchpoint to generate a Watchpoint exception on the bytes at addresses 0x2003, 0x2004 and 0x2005, program:
  - DBGWVR<n>_EL1 with 0x2000.
  - DBGWCR<n>_EL1.BAS to be 0b00111000.
- If the address programmed into the DBGWVR<n>_EL1 is doubleword-aligned:
  - To generate a Watchpoint exception when any byte in the word starting at the doubleword-aligned address is accessed, program DBGWCR<n>_EL1.BAS to be 0b00001111.
  - To generate a Watchpoint exception when any byte in the word starting at address DBGWVR<n>_EL1[31:3]:100 is accessed, program DBGWCR<n>_EL1.BAS to be 0b11110000.

Note

Arm deprecates programming a DBGWVR<n>_EL1 with an address that is not doubleword-aligned.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D2.9.4.3 Programming a watchpoint range with eight or more bytes

A debugger can use the *MASK* field, DBGWCR<n>_EL1.MASK, to program a single watchpoint with a data address range. The range must meet all of the following criteria:

- It is a size that is:
  - A power-of-two.
  - A minimum of eight bytes.
  - A maximum of 2GB.
- It starts at an address that is aligned to the size.

The MASK field specifies the number of least significant data address bits that must be masked. Up to 31 least significant bits can be masked.

If *m* least significant address bits are masked, an address match watchpoint generates a Watchpoint exception on all of the following:

- Address DBGWVR<n>_EL1[`AddrTop`:*m*]:000...0.
- Address DBGWVR<n>_EL1[`AddrTop`:*m*]:111...1.
- Any address between these two addresses.

An address mismatch breakpoint generates a Watchpoint exception for any address outside of the range.

Example D2-7

If the four least significant address bits are masked, Watchpoint exceptions are generated for all addresses between DBGWVR<n>_EL1[`AddrTop`:4]:0000 and DBGWVR<n>_EL1[`AddrTop`:4]:1111, including these addresses.

When masking address bits, a debugger must both:

- Program DBGWCR<n>_EL1.BAS to be `0b11111111`. See Programming dependencies of the BAS and MASK fields.
- In the DBGWVR<n>_EL1, set the masked address bits to 0. For watchpoint behavior when any of the masked address bits are not 0, see Other usage constraints.

D2.9.5 Watchpoint behavior for certain instruction classes

Under normal operating conditions, the following do not generate Watchpoint exceptions:

- Instruction cache maintenance instructions.
- Address translation instructions.
- TLB maintenance instructions.
- Prefetch memory instructions.
- All data cache maintenance instructions except `DC IVAC`.

However, the debug architecture allows for IMPLEMENTATION DEFINED controls, such as those in ACTLR registers, to enable watchpoints on an IMPLEMENTATION DEFINED subset of these instructions. Whether a watchpoint treats the instruction as a load or a store, and the access size of instruction cache, address translation, and TLB operations are IMPLEMENTATION DEFINED.

The access size of the IMPLEMENTATION DEFINED instruction cache, address translation, and TLB operations which generate Watchpoint exceptions are IMPLEMENTATION DEFINED.

See also the following subsections:

- Watchpoint behavior on accesses by Store-Exclusive instructions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Watchpoint behavior on accesses by the DC IVAC instruction and the DC ZVA instruction.

D2.9.5.1 Watchpoint behavior on accesses by Store-Exclusive instructions

If a watchpoint matches on a data access caused by a Store-Exclusive instruction, then:

- If the store fails because an Exclusives monitor does not permit it, it is IMPLEMENTATION DEFINED whether the watchpoint generates a Watchpoint exception.
- Otherwise, the watchpoint generates a Watchpoint exception.

D2.9.5.2 Watchpoint behavior on accesses by the DC IVAC instruction and the DC ZVA instruction

`DC ZVA` operations can generate Watchpoint exceptions.

If the Point of Coherency is before any level of cache, it is IMPLEMENTATION DEFINED whether a DC IVAC instruction can generate a Watchpoint exception. Otherwise, DC IVAC operations can generate Watchpoint exceptions.

`DC IVAC` and `DC ZVA` operations are treated as data stores by DBGWCR<n>_EL1.LSC.

Note

For the size of data accesses performed by the `DC IVAC` instruction and the `DC ZVA` instruction, see Watchpoint data address comparisons. The size of all data accesses must be considered because watchpoints can be programmed to match on individual bytes.

D2.9.6 Watchpoint usage constraints

See the following:

- Reserved DBGBCR<n>_EL1.BT values when using Watchpoints.
- Reserved DBGWCR<n>_EL1.{SSC, HMC, PAC} values.
- Usage constraints on DBGWCR<n>_EL1.LBN values.
- Programming dependencies of the BAS and MASK fields.
- Reserved DBGWCR<n>_EL1.BAS values.
- Reserved DBGWCR<n>_EL1.MASK values.
- Other usage constraints.

D2.9.6.1 Reserved DBGBCR<n>_EL1.BT values when using Watchpoints

It is CONSTRAINED UNPREDICTABLE whether Watchpoint exceptions and debug events are generated if DBGBCR<n>_EL1.BT is a reserved value. See Reserved DBGBCR<n>_EL1.BT values.

D2.9.6.2 Reserved DBGWCR<n>_EL1.{SSC, HMC, PAC} values

Table D2-12 shows when particular combinations of DBGWCR<n>_EL1.{SSC, HMC, PAC} are reserved.

Table D2-12 Reserved SSC, HMC, and PAC combinations

HMC, SSC, and PAC combination	Reserved
Combinations not included in Table D2-9.	Always.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If a watchpoint is programmed with one of these reserved combinations:

- The watchpoint must behave as if it is either:
  - Disabled.
  - Programmed with a combination that is not reserved, other than for a direct or external read of DBGWCR<n>_EL1.
- For a direct or external read of DBGWCR<n>_EL1, if the reserved combination:
  - Has no function for any execution conditions, the value read back for each of SSC, HMC, and PMC is UNKNOWN.
  - Has a function for execution conditions other than the current execution conditions, the value read back is the value written. This permits software to save and restore the combination so that the watchpoint functions for the other execution conditions.

The behavior of watchpoints with reserved combinations of SSC, HMC, and PAC might change in future revisions of the architecture. For this reason, software must not rely on the behavior described here.

D2.9.6.3 Usage constraints on DBGWCR<n>_EL1.LBN values

For Linked Watchpoints

A Linked watchpoint must link to a *context-aware* breakpoint. For a Linked watchpoint, any DBGWCR<n>_EL1.LBN value that is not for a *context-aware* breakpoint is reserved.

If a Linked watchpoint links to a breakpoint that is not implemented, or does not support linking, then reads of DBGWCR<n>_EL1.LBN return an UNKNOWN value and the behavior is CONSTRAINED UNPREDICTABLE. The Linked watchpoint behaves as if it is either:

- Disabled
- Linked to an UNKNOWN breakpoint that supports linking.

If a Linked watchpoint links to one of the following implemented breakpoints, then it behaves as if it is disabled:

- A breakpoint that is *context-aware*, but that is either not enabled or not programmed as a Context matching breakpoint with linking enabled.
- A breakpoint that supports address matching with linking and is not programmed as an Address matching breakpoint with linking enabled.

It is CONSTRAINED UNPREDICTABLE whether Watchpoint exceptions and debug events are generated if an instruction address generates a breakpoint match for more than one enabled Address matching breakpoint with linking enabled.

For Unlinked Watchpoints For Unlinked watchpoints, DBGWCR<n>_EL1.LBN reads UNKNOWN and its value is ignored.

D2.9.6.4 Programming dependencies of the BAS and MASK fields

When programming a watchpoint, a debugger must use either:

- The MASK field, to program the watchpoint with an address range that can be eight bytes to 2GB.
- The BAS field, to select which bytes in the doubleword or word starting at the address contained in the DBGWVR<n>_EL1 the watchpoint must generate Watchpoint exceptions for.

If the debugger uses the:

- MASK field, it must program BAS to be `0b11111111`, so that all bytes in the doubleword or word are selected.
- BAS field, it must program MASK to be `0b00000`, so that the MASK field does not indicate any address ranges.

If an enabled watchpoint has a MASK field that is nonzero and a BAS field that is not set to `0b11111111`, then for each byte in the address range, it is CONSTRAINED UNPREDICTABLE whether or not a Watchpoint exception is generated.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D2.9.6.5 Reserved DBGWCR<n>_EL1.BAS values

The BAS field must be programmed with a value `Zeros(8-n-m):Ones(n):Zeros(m)`, where:

- `n` is a nonzero positive integer less-than-or-equal-to 8.
- `m` is a positive integer less-than 8.
- `n+m` is less-than-or-equal-to 8.

All other values are reserved.

Note

If `x` is zero, then `Zeros(x)` is an empty bitstring.

If DBGWVR<n>_EL1[2] is 1, DBGWCR<n>_EL1.BAS[7:4] are RES0 and are ignored.

If a watchpoint is programmed with a reserved BAS value:

- It is CONSTRAINED UNPREDICTABLE whether the watchpoint generates a Watchpoint exception for each byte in the doubleword or word of memory addressed by the DBGWVR<n>_EL1.
- A direct or external read of DBGWCR<n>_EL1.BAS returns an UNKNOWN value.

Software must not rely on these properties as the behavior of reserved values might change in a future revision of the architecture.

D2.9.6.6 Reserved DBGWCR<n>_EL1.MASK values

If a watchpoint is programmed with a reserved MASK value:

- The watchpoint must behave as if it is either:
  - Disabled.
  - Programmed with an UNKNOWN value that is not reserved, that might be `0b00000`, other than for a direct or external read of DBGWCR<n>_EL1.
- A direct or external read of DBGWCR<n>_EL1.MASK returns an UNKNOWN value.

D2.9.6.7 Other usage constraints

For all watchpoints:

- DBGWVR<n>_EL1[1:0] are RES0 and are ignored.
- If DBGWCR<n>_EL1.MASK is nonzero, and any masked bits of DBGWVR<n>_EL1 are not 0, it is CONSTRAINED UNPREDICTABLE whether the watchpoint generates a Watchpoint exception when the unmasked bits match.
- A watchpoint never generates any Watchpoint exceptions if DBGWCR<n>_EL1.LSC is `0b00`.

D2.9.7 Exception syndrome information, fault address information, and preferred return address

On taking a Watchpoint exception or entering Debug state for a Watchpoint debug event, the PE records the following:

- Information about the debug event or exception in a *syndrome register*. No information is recorded if the watchpoint causes the PE to enter Debug state.
- An address that the debugger can use to determine the memory location that caused the watchpoint, in a *Fault Address Register*. No address is recorded if the PE sets the FnV bit in the syndrome register to 1, as described in Fault address information.

The syndrome register and fault address register used are one of the following:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- For a Watchpoint exception taken to EL1, ESR_EL1 and FAR_EL1.
- For a Watchpoint exception taken to EL2, ESR_EL2 and FAR_EL2.
- For a Watchpoint debug event that causes entry to Debug state, EDHSR, if implemented, and EDWAR.

In cases where one instruction triggers multiple watchpoints, only one address is recorded.

See ISS encoding for an exception from a Watchpoint exception for more information.

This section uses the following terms:

Naturally-aligned block of memory

Means:

- A power-of-two size.
- No larger than the `DC ZVA` block size defined by the DCZID_EL0.BS field, if ESR_ELx.FnP or EDHSR.FnP is 0.
- No larger than the smallest implemented translation granule if ESR_ELx.FnP or EDHSR.FnP is 1.
- Contains a watchpointed address accessed by the memory access or set of contiguous memory accesses that triggered the watchpoint.

The size of the naturally-aligned block of memory is IMPLEMENTATION DEFINED. There is no architectural means of discovering the size.

Watchpointed address

Is defined as any address for which at least one of the address match watchpoints generates a watchpoint match.

For more information, see the subsections that follow. These are:

- Exception syndrome information.
- Fault address information.
- Preferred return address.

D2.9.7.1 Exception syndrome information

R_{KSSHC} If a watchpoint matches an access that is due to a *relaxed watchpoint access*, then the PE:

- Sets ESR_ELx.FnV to an IMPLEMENTATION DEFINED value, 0 or 1, on taking a Watchpoint exception generated by the watchpoint match.
- If FEAT_EDHSR is implemented, sets EDHSR.FnV to an IMPLEMENTATION DEFINED value, 0 or 1, on entering Debug state on a Watchpoint debug event generated by the watchpoint match.

Otherwise, ESR_ELx.FnV or EDHSR.FnV is set to 0.

R_{RLWSF} When the PE sets ESR_ELx.FnV to 0 on taking a Watchpoint exception generated by a watchpoint match:

- If the lowest watchpointed address that is higher than or equal to the address recorded in FAR_ELx might not have been accessed by the instruction, other than as permitted by R_{LVLYV}, the PE sets ESR_ELx.FnP to 1.
- Otherwise, the PE sets ESR_ELx.FnP to 0.

R_{TLMWP} When FEAT_Debugv8p2 is implemented, the PE behavior on taking a Watchpoint exception is as follows:

- If the PE sets ESR_ELx.FnV to 1 or ESR_ELx.FnP to 1, then the PE sets ESR_ELx.WPTV to 1.
- If the address recorded in FAR_ELx is not within a naturally-aligned block of memory, then the PE sets ESR_ELx.WPTV to 1.
- Otherwise, the PE sets ESR_ELx.WPTV to an IMPLEMENTATION DEFINED value, 0 or 1.

On a watchpoint match generated by watchpoint <*n*>:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the PE sets ESR_ELx.WPTV to 1 on taking a Watchpoint exception generated by the watchpoint match, then ESR_ELx.WPT is set to <*n*>.
- If FEAT_EDHSR is implemented and the PE sets EDHSR.WPTV to 1 on entering Debug state on a Watchpoint debug event generated by the watchpoint match, then EDHSR.WPT is set to <*n*>.
- Otherwise, ESR_ELx.WPT or EDHSR.WPT is UNKNOWN.

R_{XQBQN} When an instruction generates multiple watchpoint matches and the PE sets ESR_ELx.WPTV to 1 on taking a Watchpoint exception generated by the watchpoint match,or on entering Debug state on a Watchpoint debug event generated by the watchpoint match, then the value <n> reported in ESR_ELx.WPT is defined as follows:

- If all the matching watchpoints are address match watchpoints or all the matching watchpoints are address mismatch watchpoints, then <n> is any watchpoint that generates a watchpoint match.
- Otherwise, <n> is any address match watchpoint that generates a watchpoint match.

D2.9.7.2 Fault address information

D2.9.7.2.1 Address recorded for Watchpoint exceptions generated by zeroing and data cache maintenance instructions

R_{GBDMM} For a watchpoint triggered by a DC ZVA instruction, or a Data Cache maintenance operation, the address recorded in FAR_ELx or EDWAR is both:

- Any address accessed by the instruction.
- Within a naturally-aligned block of memory.

Note

Despite the mnemonic, the DC ZVA instruction is not a data cache maintenance instruction.

D2.9.7.2.2 Address recorded for Watchpoint exceptions generated by other instructions

The address recorded by other instructions must be both:

- From the inclusive range between:
  - The lowest address accessed by the memory access or set of contiguous memory accesses that triggered the watchpoint.
  - The highest watchpointed address accessed by the memory access or set of contiguous memory accesses that triggered the watchpoint.
- Within a naturally-aligned block of memory.

Example D2-8 Address recorded for a watchpoint programmed on `0x8019`

A debugger programs a watchpoint to generate a Watchpoint exception on any access to the byte `0x8019`.
An A32 load multiple instruction then loads nine registers starting from address `0x8004` upwards. This triggers the watchpoint.
If the `DC ZVA` block size is:

- 32 bytes, the address that the PE records must be between `0x8004` and `0x8019` inclusive.
- 16 bytes, the address that the PE records must be between `0x8010` and `0x8019` inclusive.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D2.9.7.3 Preferred return address

The preferred return address of a Watchpoint exception is the address of the instruction that was not executed because the PE took the Watchpoint exception instead.

This means that the preferred return address is the address of the instruction that caused the exception.

D2.9.8 Pseudocode description of Watchpoint exceptions taken from AArch64 state

`AArch64.WatchpointByteMatch()` tests an individual byte accessed by an operation.

`AArch64.StateMatch()` tests the values in DBGWCR<n>_EL1.{HMC, SSCE, SSC, PAC}, and if the watchpoint links to a breakpoint with linking enabled, also tests the breakpoint that the watchpoint links to.

`AArch64.WatchpointMatch()` tests the value in DBGWVR<n>_EL1.

`AArch64.CheckWatchpoint()` generates a `FaultRecord` that `AArch64.Abort()` raises a Watchpoint exception for if all of the following are true:

- MDSCR_EL1.MDE is 1.
- Debug exceptions are enabled from the current Exception level and Security state. See Enabling debug exceptions from the current Exception level and Security state.
- All of the conditions required for Watchpoint exception generation are met. See About Watchpoint exceptions.

Note

`AArch64.CheckWatchpoint()` might halt the PE and cause it to enter Debug state. External debug uses Debug state.

`AArch64.WatchpointException()` is called to generate a Watchpoint exception.

These functions are defined in Pseudocode for AArch64 operation.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D2.10 Software Step exceptions

The following subsections describe Software Step exceptions:

- About Software Step exceptions.
- Rules for setting MDSCR_EL1.SS to 1.
- The software step state machine.
- Entering the active-not-pending state.
- Behavior in the active-not-pending state.
- Entering the active-pending state.
- Behavior in the active-pending state.
- Exception syndrome information and preferred return address.
- Additional considerations.
- Pseudocode description of Software Step exceptions.

D2.10.1 About Software Step exceptions

Software step is a resource that a debugger can use to make the PE single-step instructions.

For example, by using software step, debugger software executing at a higher Exception level can single-step instructions at a lower Exception level.

Operation is as follows:

1. A debugger:
   a. Enables software step by setting MDSCR_EL1.SS to 1. See The debug exception enable controls.
   b. Executes an exception return instruction, to branch to the instruction to be single-stepped in the software being debugged.
   c. Optionally configures the MDSTEPOP_EL1 register. See Behavior in the active-not-pending state.
2. The PE then:
   a. Executes the instruction to be single-stepped.
   b. Takes a Software Step exception on the next instruction, returning control to the debugger.

However, another exception might be generated while the instruction is being stepped. This exception is either:

- A synchronous exception that is generated by the instruction being stepped.
- An asynchronous exception that is taken before or after the instruction being stepped.

The PE can take a Software Step exception only if debug exceptions are enabled from the current Exception level and Security state. See Enabling debug exceptions from the current Exception level and Security state.

A state machine describes the behavior of software step, shown in The software step state machine.

Throughout this Software Step exceptions section, including in all subsections, EL_D means the Exception level that Software Step exceptions are targeting. Routing debug exceptions defines EL_D as the *debug target Exception level*.

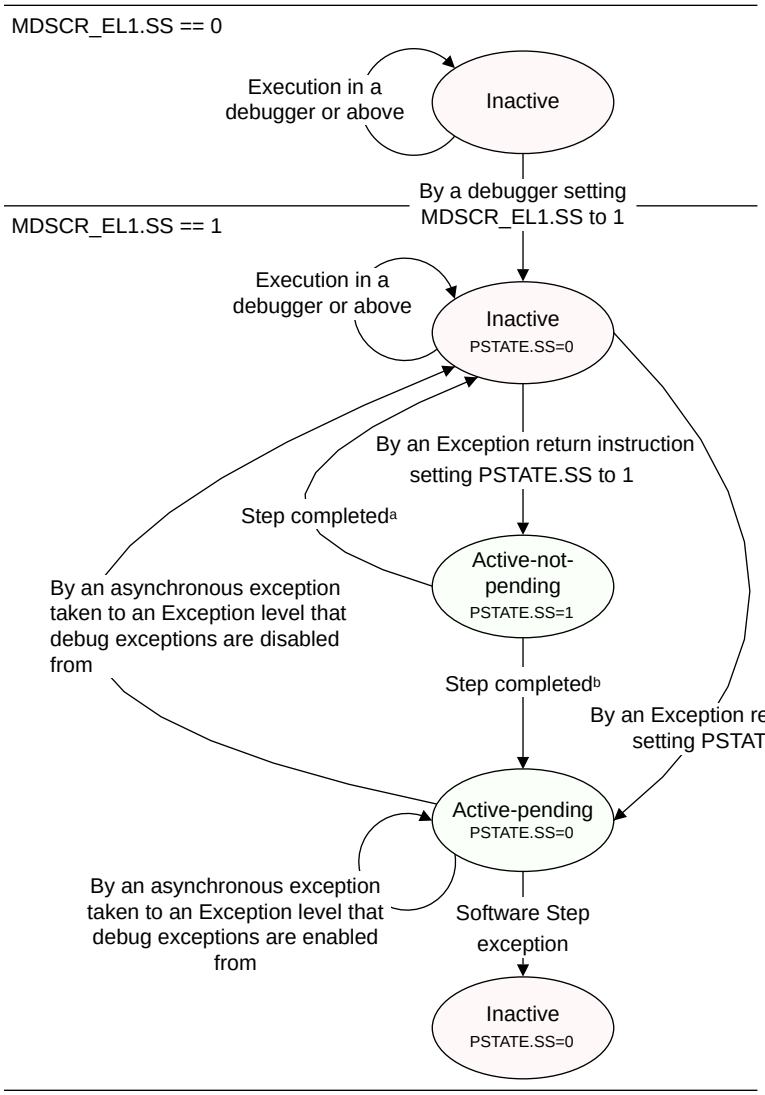
D2.10.2 Rules for setting MDSCR_EL1.SS to 1

Debugger software must be executing in an Exception level and Security state that debug exceptions are disabled from when it sets MDSCR_EL1.SS to 1.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D2.10.3 The software step state machine

In Figure D2-3, the OS Lock is unlocked and `DoubleLockStatus()` == FALSE.



Execution is at either:

- An Exception level that is higher than EL_D.
- EL_D with (PSTATE.D == 1 || MDSCR_EL1.KDE == 0).

This is termed *execution in a debugger or above*.

To make the PE single-step an instruction, the debugger:

1. Sets SPSR_ELx.SS to 1.
2. Programs the ELR_ELx to point to the instruction to be stepped.
3. Executes an Exception return instruction.

Execution is in the software being debugged, at either:

- An Exception level that is lower than EL_D.
- EL_D with (PSTATE.D == 0 && MDSCR_EL1.KDE == 1).

Execution is in the software being debugged, at either:

- An Exception level that is lower than EL_D.
- EL_D with (PSTATE.D == 0 && MDSCR_EL1.KDE == 1).

A Software Step exception is pending.

Execution has returned to the debugger.

a. The step is the PE either:
   - Taking an exception to an Exception level that debug exceptions are disabled from.
   - If execution is at EL_D with MDSCR_EL1.KDE == 1, executing an instruction that sets PSTATE.D to 1.

   Software step is inactive when debug exceptions are disabled from the current Exception level, and debug exceptions are disabled from EL_D when PSTATE.D is 1.

b. The step is the PE either:
   - Executing the instruction to be stepped without taking an exception.
   - Taking an exception to an Exception level that debug exceptions are enabled from. The Exception level might be using AArch64 or AArch32.

c. Or, if execution is at EL_D with MDSCR_EL1.KDE == 1, by software setting PSTATE.D to 0.

Figure D2-3 Software step state machine

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For a description of when debug exceptions are enabled or disabled from an Exception level, see Enabling debug exceptions from the current Exception level and Security state.

For more information about how a step is completed, see Behavior in the active-not-pending state.

The software step states are:

Inactive Software step is inactive. It cannot generate any Software Step exceptions or affect PE execution. Software step is inactive whenever any of the following are true:

- Debug exceptions are disabled from the current Exception level.
- MDSCR_EL1.SS is 0.

Active-not-pending

None of the conditions mentioned in Inactive are true, therefore software step is active.

The instruction to be stepped is usually the instruction in memory at the address specified by the PC.

Active-pending

None of the conditions mentioned in Inactive are true, therefore software step is active.

A Software Step exception is pending on the current instruction.

Whenever software step is active, whether the state machine is in the active-not pending state or the active-pending state depends on PSTATE.SS. Table D2-13 shows this.

Table D2-13 State machine states

Debug exception enable status in the current Exception level and Security state	MDSCR_EL1.SS	PSTATE.SS	State machine state
Disabled	X	X	Inactive
Enabled	0	X	Inactive
Enabled	1	1	Active-not-pending
Enabled	1	0	Active-pending

D2.10.4 Entering the active-not-pending state

Software step can only enter the active-not-pending state from the inactive state.

Software step:

- Enters the active-not-pending state when an Exception return instruction writes 1 to PSTATE.SS, by copying from SPSR_ELx.SS when it restores PSTATE.
- Might enter the active-not-pending state on exiting Debug state when DSPSR_EL0.SS is 1. See Exiting Debug state.

An Exception return instruction only copies 1 from SPSR_ELx.SS to PSTATE.SS if all of the following are true:

- MDSCR_EL1.SS is 1.
- Debug exceptions are disabled from the current Exception level.
- Debug exceptions are enabled from the Exception level that the Exception return instruction targets.

Otherwise, Exception return instructions set PSTATE.SS to 0, regardless of the value of SPSR_ELx.SS.

Table D2-14 shows this. In the table:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Lock Means the value of (OSLSR_EL1.OSLK == '1' || `DoubleLockStatus()`).

TGE Means the value of HCR_EL2.TGE. If EL2 is not implemented, the PE behaves as if this is 0.

TDE Means the Effective value of MDCR_EL2.TDE. See Routing debug exceptions.

Table D2-14 Value an Exception return instruction writes to PSTATE.SS

MDSCR_EL1.SS	Lock	TGE	TDE	Value an Exception return instruction writes to PSTATE.SS
0	X	X	X	0
1	TRUE	X	X	0
	FALSE	X	X	See Table D2-15
		0	0	See Table D2-15
			1	See Table D2-16
		1	X	See Table D2-16

For:

- If EL_D is EL1 using AArch64, Table D2-15 shows the value an Exception return instruction writes to PSTATE.SS.
- If EL_D is EL2 using AArch64, Table D2-16 shows the value an Exception return instruction writes to PSTATE.SS.

In both tables:

From EL Means the Exception level at which the PE executes the Exception return instruction.

Target EL Is the target Exception level of the Exception return instruction.

Note

If the Exception return instruction is an illegal exception return, the target Exception level of the Exception return instruction is the current Exception level. See Illegal exception returns from AArch64 state.

KDE Is MDSCR_EL1.KDE. See Enabling debug exceptions from the current Exception level and Security state

Table D2-15 Value an Exception return instruction writes to PSTATE.SS if EL_D is EL1 using AArch64

					Software step enable status at:		
From EL	Target EL	KDE	PSTATE.D	SPSR_ELx.D	From EL	Target EL	Value an Exception return instruction writes to PSTATE.SS
EL2	EL2	X	X	X	Disabled	Disabled	0
	EL1	0	X	X	Disabled	Disabled	0
		1	X	1	Disabled	Disabled	0
				0	Disabled	Enabled	SPSR_EL2.SS
	EL0	X	X	X	Disabled	Enabled	SPSR_EL2.SS

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

From EL	Target EL	KDE	PSTATE.D	SPSR_ELx.D	Software step enable status at: From EL	Software step enable status at: Target EL	Value an Exception return instruction writes to PSTATE.SS
EL1	EL1	0	X	X	Disabled	Disabled	0
		1	0	X	Enabled[a]	-[b]	0
			1	1	Disabled	Disabled	0
				0	Disabled	Enabled	SPSR_EL1.SS
	EL0	0	X	X	Disabled	Enabled	SPSR_EL1.SS
		1	0	X	Enabled[a]	Enabled	0
			1	X	Disabled	Enabled	SPSR_EL1.SS

a. Because MDSCR_EL1.SS == 1, it means that the Exception return instruction is itself being stepped.

b. Depends on SPSR_EL1.D.

Table D2-16 Value an Exception return instruction writes to PSTATE.SS if EL_D is EL2 using AArch64

From EL	Target EL	KDE	PSTATE.D	SPSR_ELx.D	Software step enable status at: From EL	Software step enable status at: Target EL	Value an Exception return instruction writes to PSTATE.SS
EL2	EL2	0	X	X	Disabled	Disabled	0
		1	0	X	Enabled[a]	-[b]	0
			1	1	Disabled	Disabled	0
				0	Disabled	Enabled	SPSR_EL2.SS
	EL1	0	X	X	Disabled	Enabled	SPSR_EL2.SS
		1	0	X	Enabled[a]	Enabled	0
			1	X	Disabled	Enabled	SPSR_EL2.SS
	EL0	0	X	X	Disabled	Enabled	SPSR_EL2.SS
		1	0	X	Enabled[a]	Enabled	0
			1	X	Disabled	Enabled	SPSR_EL2.SS
EL1	EL1	X	X	X	Enabled[a]	Enabled	0
	EL0	X	X	X	Enabled[a]	Enabled	0

a. Because MDSCR_EL1.SS == 1, it means that the Exception return instruction is itself being stepped.

b. Depends on SPSR_EL2.D.

D2.10.5 Behavior in the active-not-pending state

In this state, the PE does one of the following:

- Executes the instruction to be stepped and either:
  - Completes it without taking a synchronous exception.
  - Takes a synchronous exception if the instruction generates one.
- Takes an asynchronous exception without executing any instructions.
- Enters Debug state because of a *Halting debug event*.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The instruction to be stepped is usually the instruction in memory at the address specified by the PC.

If the PE executes the instruction without taking any exceptions, then the PE sets PSTATE.SS to 0, meaning that after the instruction has been executed:

- If the instruction has disabled debug by setting PSTATE.D to 1 then software step advances to the inactive state.
- If the instruction disables software step by a direct write to a System register, for example a write to MDSCR_EL1.KDE or MDSCR_EL1.SS, then software step might advance to the inactive state. These writes require explicit synchronization to guarantee their effect. See Synchronization and the software step state machine.
- Otherwise, software step advances to the active-pending state. See Behavior in the active-pending state.

If the PE takes either a synchronous or an asynchronous exception, behavior is as described in one of the following:

- If the PE takes an exception to an Exception level that is using AArch64.

If the PE enters Debug state because of a Halting debug event, behavior is as described in Entering Debug state and Software Step.

D2.10.5.1 If the PE takes an exception to an Exception level that is using AArch64

As part of exception entry, the PE does all of the following:

- Sets SPSR_ELx.SS to 0 or 1, depending on the exception. See Table D2-17.
- It is UNPREDICTABLE whether SPSR_ELx.SS to 0 or 1 when an SError exception is taken to ELx without executing the instruction.
- Sets PSTATE.SS to 0. This causes software step to enter either the active-pending state or the inactive state, depending on whether debug exceptions are enabled or disabled from the Exception level that the exception is taken to:

  Enabled Software step enters the active-pending state.

  Disabled Software step enters the inactive state.

  In either case, on taking the exception, a step is complete.
- Sets PSTATE.D to 1.

Table D2-17 Categorization of exceptions, for setting SPSR_ELx.SS to 0 or 1

Exception description	Exceptions	SPSR_ELx.SS
Exceptions whose preferred return address is for the instruction that follows the instruction to be stepped.	Supervisor Call (SVC) exceptions. Hypervisor Call (HVC) exceptions. Secure Monitor Call (SMC) exceptions.	0
Exceptions whose preferred return address is the address of the instruction to be stepped.	All other synchronous exceptions, and asynchronous exceptions that are taken before the instruction to be stepped.	1

Note

If an SMC instruction executed at Non-secure EL1 is trapped to EL2 because HCR_EL2.TSC is 1, the exception is a Trap exception, not a Secure Monitor Call exception, and so SPSR_ELx.SS is set to 1, not 0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D2.10.5.2 Summary of behavior in the active-not-pending state

Table D2-18 summarizes behavior in the active-not-pending state.

Table D2-18 Summary of behavior in the active-not-pending state

Event	Value written to PSTATE.SS	Details[a]	Value written to SPSR_ELx.SS	Next state
No exception	0	Disables Software step	n/a	Inactive
		Otherwise	n/a	Active-pending
Exception	0	Supervisor Call (SVC) Hypervisor Call (HVC)	0	Active-pending or inactive[b]
		Other	1	

a. For the *No exception* rows, this column shows the effect of the event.

For the *Exception* rows, this column shows the exception taken.

b. Which state software step enters depends on whether debug exceptions are enabled or disabled from the target Exception level. See Figure D2-3.

D2.10.6 Entering the active-pending state

Software step enters the active-pending state after any of the following operations, provided that both:

- MDSCR_EL1.SS is 1.
- Debug exceptions are enabled from the Exception level and Security state that execution is in after the operation.

The operations are:

While software step is in the active-not-pending state

The PE either:

- Executing the instruction to be stepped without taking any exceptions.
- Taking an exception.

While software step is in the active-pending state

The PE takes an asynchronous exception.

While software step is in the inactive state

The PE executes either:

- An Exception return instruction when SPSR_ELx.SS is 0.
- An instruction that enables debug by setting PSTATE.D to 0.

Note

If entry to the active-pending state is because of the PE taking an exception, it means that the exception is one that is taken to EL1 when MDCR_EL2.TDE is 1 and EL2 is implemented and enabled in the current Security state. Otherwise, debug exceptions are masked by PSTATE.D, therefore they would be disabled from the target Exception level of the exception.

In addition, software step might enter the active-pending state either:

- After a direct write to a System register, for example a write to MDSCR_EL1.KDE or MDSCR_EL1.SS. These writes require explicit synchronization to guarantee their effect. See Synchronization and the software step state machine.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On exiting Debug state when DSPSR_EL0.SS is 0. See Exiting Debug state.

D2.10.7 Behavior in the active-pending state

When the PE is in the active-pending state, a Software Step exception is taken before the PE executes an instruction.

The Software Step exception has higher priority than all other types of synchronous exception. However, the prioritization of this exception with respect to any unmasked pending asynchronous exception is not defined by the architecture.

For more information, see the following:

- Prioritization of Synchronous exceptions taken to AArch64 state.
- Prioritization of interrupts.
- Architectural requirements for taking asynchronous exceptions.

D2.10.8 Exception syndrome information and preferred return address

See the following:

- Exception syndrome information.
- Preferred return address.

D2.10.8.1 Exception syndrome information

On taking a Software Step exception, the PE records information about the exception in the Exception Syndrome Register (ESR_ELx) at the Exception level the exception is taken to. See ISS encoding for an exception from a Software Step exception for more information.

If no instruction was stepped because software step entered the active-pending state from the inactive state without passing through the active-not-pending state, then ESR_ELx.{ISV, EX} are set to {0, 0}.

When an instruction has been stepped, if the stepped instruction was a conditional Load-Exclusive instruction that failed its Condition code test, then ESR_ELx.ISV is set to 1 and ESR_ELx.EX is set to a CONSTRAINED UNPREDICTABLE choice of 0 or 1.

When an instruction has been stepped, if the stepped instruction was an Exception return instruction or an `ISB`. then ESR_ELx.ISV is set to a CONSTRAINED UNPREDICTABLE choice of 0 or 1, and ESR_ELx.EX is set to 0.

If the PE is executing at EL1 or EL0, and the Effective value of MDCR_EL2.TDE is 1, then a different exception might be taken to EL1 before the Software Step exception is taken to EL2. In this case, on taking the Software Step exception ESR_EL2.ISV is set to a CONSTRAINED UNPREDICTABLE choice of 0 or 1, and:

- If ESR_EL2.ISV is set to 1, then ESR_EL2.EX is set to the correct value for the instruction.
- If ESR_EL2.ISV is set to 0, then ESR_EL2.EX is set to 0.

Other than for the cases described above, when an instruction has been stepped:

- ESR_ELx.ISV is set to 1, to indicate that ESR_ELx.EX is valid.
- If the instruction stepped was a Load-Exclusive instruction, then ESR_ELx.EX is set to 1. Otherwise, ESR_ELx.EX is set to 0.

Note

If the PE cannot determine the correct value of ESR_ELx.EX for the stepped instruction, then it sets ESR_ELx.{ISV, EX} to {0, 0}. For example, the exception is taken before the PE decodes the stepped instruction, or the exception means the PE has no valid stepped instruction to decode.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

A Load-Exclusive instruction is any instruction that has a mnemonic starting with either `LDX` or `LDAX`.

Note

An implementation that always sets ISV to 0 and never sets EX is not compliant.

Table D2-19 summarizes the possible values that the PE can record in ESR_ELx.{ISV, EX}.

Table D2-19 Values that the PE can record in ESR_ELx.{ISV, EX}

Description	ESR_ELx.ISV	ESR_ELx.EX
Syndrome data is not available because no instruction was stepped.	0	0
Syndrome data is available because an instruction was stepped. The instruction stepped was a conditional Load-Exclusive instruction that failed its Condition code test.	1	0 or 1
Syndrome data is available because an instruction was stepped. The instruction stepped was an Exception Return instruction or an `ISB`.	0 or 1	0
A different exception is taken before the Software Step exception.	0	0
	1	Set to the correct value for the instruction.
Syndrome data is available because an instruction was stepped. The instruction stepped was an instruction other than a Load-Exclusive instruction, an Exception Return instruction, or an `ISB`, and no other exception was taken before the Software Step exception.	1	0
Syndrome data is available because an instruction was stepped. The instruction stepped was a Load-Exclusive instruction that was either not conditional or did not fail its Condition code test.	1	1

D2.10.8.2 Preferred return address

The preferred return of a Software Step exception is the address of the instruction that was not executed because the PE took the Software Step exception instead.

D2.10.9 Additional considerations

This section contains the following:

- Behavior when an Exception return instruction is an illegal exception return.
- Behavior when the instruction stepped writes a misaligned PC value.
- Stepping code that uses Exclusives monitors.
- Synchronization and the software step state machine.

D2.10.9.1 Behavior when an Exception return instruction is an illegal exception return

If the conditions for entering the active-not-pending state in Entering the active-not-pending state are met, but the PE executes an Exception return instruction that is an illegal exception return, the exception return must be taken to the same Exception level that it was taken from. In this scenario, even though the Exception level remains the same before and after the Exception return instruction, software step can advance from the inactive state to one of the active states. Consider the following case:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

1. MDSCR_EL1.SS is 1 and software step is inactive. The current Exception level is EL1 using AArch64, the OS Lock and OS Double Lock are unlocked, and MDCR_EL2.TDE is 0, MDSCR_EL1.KDE is 1, and PSTATE.D is 1.

   PSTATE.D == 1 is the reason why software step is inactive, because PSTATE.D == 1 means that debug exceptions are disabled from the current Exception level.
2. The PE executes an Exception return instruction.
3. The intended target of the Exception return instruction is EL2. This means that the Exception return instruction is an illegal exception return because the intended target is higher than the Exception level the Exception return instruction it is executed at. In this case, the Exception return instruction must target EL1 instead of EL2.

   If SPSR_EL1.D is 0, then on the Exception return instruction PSTATE.D becomes 0 and debug exceptions become enabled from the current Exception level. Software step therefore advances from the inactive state to one of the active states.

Which active state software step advances to depends on whether SPSR_ELx.SS is 1 or 0:

- If SPSR_ELx.SS is 1, software step advances to the active-not-pending state.

  In this case, an Illegal Execution state exception is pending on the instruction to be stepped, and the PE takes the Illegal Execution state exception instead of executing the instruction to be stepped.
- If SPSR_ELx.SS is 0, software step advances to the active-pending state.

  In this case, a Software Step exception and an Illegal Execution state exception are both pending. The Software Step exception has higher priority. On taking the Software Step exception, the PE sets SPSR_ELx.IL to 1.

Note

Prioritization of Synchronous exceptions taken to AArch64 state shows the relative priorities of synchronous exceptions.

D2.10.9.2 Behavior when the instruction stepped writes a misaligned PC value

An indirect branch that writes a misaligned PC value might generate a PC alignment fault exception at the target of the branch. However, if the indirect branch is stepped using software step, the PE takes a Software Step exception instead, because the Software Step exception has higher priority. Behavior on returning from the Software Step exception depends on which Execution state the Exception level being returned to is using:

AArch64 A PC alignment fault exception is generated.

D2.10.9.3 Stepping code that uses Exclusives monitors

The A-profile architecture provides no mechanism for preserving the state of the Exclusives monitors when a Load-Exclusive or a Store-Exclusive instruction is stepped.

However, for certain progressions through the software step state machine, on taking a Software Step exception, the PE provides an indication of whether the instruction stepped was a Load-Exclusive instruction.

Debugger software can use this to detect the state of the Exclusives monitors. For example, if the PE reports that the instruction stepped was a Load-Exclusive instruction, the debugger is aware that the next Store-Exclusive operation will fail, because all Exclusives monitors are cleared on returning from the Software Step exception. The debugger must then take action to ensure that the code being stepped makes forwards progress.

For more information on how the PE reports whether the instruction stepped was a Load-Exclusive instruction, see Exception syndrome information and preferred return address.

D2.10.9.4 Synchronization and the software step state machine

Any of the following can cause transitions between software step states:

- A direct write to a System register.
- A direct write to a Special-purpose register.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- A write to an external debug register.

The software step state machine indirectly reads some of these registers and so is not guaranteed to observe any new values until after a Context Synchronization event has occurred.

Between a write to the register and the next Context Synchronization event, it is CONSTRAINED UNPREDICTABLE whether software step uses the state of the PE before the write, or the state of the PE after the write.

After a Context Synchronization event, the state machine must use the state of the PE after the write.

Example D2-9 Example of synchronization and software step state machine changing states

1. Software changes MDSCR_EL1.SS from 0 to 1 when debug exceptions are enabled.
2. The PE executes some instructions.
3. A Context Synchronization event occurs.

During step 2, it is CONSTRAINED UNPREDICTABLE whether software step remains in the inactive state, as if MDSCR_EL1.SS is 0, or enters the active-pending state because MDSCR_EL1.SS is 1. If it is in the:

- Inactive state, then after the Context Synchronization event, it must enter the active-pending state.
- Active-pending state, the PE might take a Software Step exception before the Context Synchronization event.

Note

A direct write to a Special-purpose register does not require explicit synchronization.

D2.10.10 Pseudocode description of Software Step exceptions

`SSAdvance()` advances software step from the active-not-pending state to the active-pending state, by setting PSTATE.SS to 0. It is called on completing execution of each instruction.

`CheckSoftwareStep()` checks whether software step is in the active-pending state, and if it is, generates a Software Step exception. It is called before each instruction executed, regardless of Execution state, before checking for any other synchronous exceptions.

`DebugExceptionReturnSS()` returns the value to write to PSTATE.SS on an exception return or an exit from Debug state. See Entering the active-not-pending state.

These functions are defined in R-profile Architecture Pseudocode.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D2.11 Synchronization and debug exceptions

The behavior of debug depends on all of the following:

- The state of the external debug authentication interface.
- Indirect reads of:
  - External debug registers.
  - System registers, including system debug registers.
  - Special-purpose registers.

If a change is made to any of these, the effect of that change on debug exception generation cannot be relied on until after a Context Synchronization event has occurred. Similarly, the effect of the change on the software step state machine cannot be relied on until after a Context Synchronization event has occurred.

For any instructions executed between the time when the change is made and the time when the next Context Synchronization event occurs, it is CONSTRAINED UNPREDICTABLE whether debug uses the state of the PE before the change, or the state of the PE after the change.

Example D2-10 Example of synchronization and Breakpoint exception generation

1. Software changes MDSCR_EL1.MDE from 0 to 1.
2. An instruction is executed, that would cause a Breakpoint exception if self-hosted debug uses the state of the PE after the change.
3. A Context Synchronization event occurs.

In this case, it is CONSTRAINED UNPREDICTABLE whether the instruction generates a Breakpoint exception.

Example D2-11 Example of synchronization and debug exceptions generation

1. Software unlocks the OS Lock.
2. The PE executes some instructions.
3. A Context Synchronization event occurs.

During the time when the PE is executing some instructions, step 2, it is CONSTRAINED UNPREDICTABLE whether debug exceptions other than Breakpoint Instruction exceptions can be generated.

Note

For more information, see:

- Accessing PSTATE fields.
- Synchronization requirements for AArch64 System registers.
- Synchronization of changes to the external debug registers.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter D3
AArch64 Self-hosted Trace

This chapter describes the AArch64 self-hosted trace:

Introductory information:

- About self-hosted trace.
- Trace sinks.
- Register controls to enable self-hosted trace.

Prohibited regions in trace:

- Controls to prohibit trace at Exception levels.
- Self-hosted trace and visibility of virtual data.

Timestamps and Synchronization:

- Self-hosted trace timestamps.
- Synchronization in self-hosted trace.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D3.1 About self-hosted trace

A trace unit generates trace data to describe the program flow of the PE.

The trace unit can be an implementation of any of the following:

- In Armv8, a standard Arm Embedded Trace Macrocell (ETM). See *Arm® Embedded Trace Macrocell Architecture Specification, ETMv4*.
- An IMPLEMENTATION DEFINED trace function.

If an Armv8.4-compliant PE implements an ETM Architecture trace unit that includes the ETM System register interface, FEAT_TRF must be implemented.

If any of the following are true, then Arm recommends that FEAT_TRF is implemented, but this is not mandatory:

- An Armv8.4-compliant PE implements an IMPLEMENTATION DEFINED trace unit.
- An Armv8.4-compliant PE implements an ETM Architecture trace unit but does not implement the ETM System register interface.

Self-hosted trace happens when the agent controlling the trace collection is part of the same software stack as the software being traced. The agent controls prohibited regions. The information collected by the agent is sent to a trace sink.

The trace unit and the PE must have the same view of the debug authentication interface. If FEAT_TRF is implemented, `ExternalNoninvasiveDebugEnabled`() is always TRUE.

D3.1.1 Trace sinks

The trace unit sends the trace data to a trace sink. A system might include multiple trace sinks, and allow software to configure which trace sink or sinks are used.

An example of an internal trace sink is an Embedded Trace Router (ETR), which allows software to define a buffer in memory. Trace data is written to this buffer.

In Armv8, Arm recommends that a system that includes FEAT_TRF incorporates an ETR, and follows the system architecture described by the *CoreSight Base System Architecture (CS-BSA)*.

D3.1.2 Register controls to enable self-hosted trace

If FEAT_TRF is implemented, self-hosted trace is enabled if any of the following are true:

- EDSCR.TFO == 0.
- EDSCR.TFO == 1, and `ExternalSecureNoninvasiveDebugEnabled`() == FALSE.
- EDSCR.TFO ==1, the PE executes in Secure state, and `ExternalSecureNoninvasiveDebugEnabled`() == FALSE.
- EDSCR.TFO ==1, and `ExternalRealmNoninvasiveDebugEnabled`() == FALSE.

The pseudocode function `SelfHostedTraceEnabled`() shows these rules.

If FEAT_TRF is not implemented, `SelfHostedTraceEnabled`() returns FALSE.

While `SelfHostedTraceEnabled`() is FALSE, `ExternalNoninvasiveDebugAllowed`() controls whether tracing is prohibited or allowed in each Security state.

The self-hosted trace extensions do not provide any mechanism to control software access to the trace unit external debug interface.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D3.2 Prohibited regions in self-hosted trace

Trace is not generated in prohibited regions. The pseudocode function `TraceAllowed`() indicates whether tracing is allowed in the current Security state and Exception level.

The IMPLEMENTATION DEFINED debug authentication interface can allow an external agent to disable the self-hosted trace extension.

D3.2.1 Controls to prohibit trace at Exception levels

If `SelfHostedTraceEnabled`() == TRUE, TRFCR_EL1 and TRFCR_EL2 control whether trace is prohibited at an Exception level. While `SelfHostedTraceEnabled`() == FALSE, the registers TRFCR_EL1 and TRFCR_EL2 are ignored.

If `SelfHostedTraceEnabled`() == TRUE, tracing is prohibited at EL0 if one of the following is true:

- The Effective value of HCR_EL2.TGE == 0 and TRFCR_EL1.E0TRE == 0.
- The Effective value of HCR_EL2.TGE == 1 and TRFCR_EL2.E0HTRE == 0.

If `SelfHostedTraceEnabled`() == TRUE, tracing is prohibited at EL1 if TRFCR_EL1.E1TRE == 0.

If `SelfHostedTraceEnabled`() == TRUE, tracing is prohibited at EL2 if TRFCR_EL2.E2TRE == 0.

The pseudocode `TraceAllowed()` shows the above rules.

When PMCR_EL0.X==0, no PMU events are exported to the trace unit.

PMU events are exported to the trace unit.

If `SelfHostedTraceEnabled`() == TRUE, Table D3-1 shows the prohibited regions by Exception level and state.

In the table:

TGE Means the Effective value of HCR_EL2.TGE.

P Means prohibited.

E2TRE Means prohibited if TRFCR_EL2.E2TRE == 0.

E1TRE Means prohibited if TRFCR_EL1.E1TRE == 0.

E0HTRE Means prohibited if TRFCR_EL2.E0HTRE == 0.

E0TRE Means prohibited if TRFCR_EL1.E0TRE == 0.

n/a Not applicable.

If FEAT_TRBE is not implemented, trace is allowed unless otherwise prohibited in Table D3-1.

Table D3-1 Prohibited regions

Controls		Tracing prohibited at		
State	TGE	EL2	EL1	EL0
Non-secure	0	E2TRE	E1TRE	E0TRE
	1	E2TRE	n/a	E0HTRE
Secure	0	E2TRE	E1TRE	E0TRE
	1	E2TRE	n/a	E0HTRE

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D3.2.2 Self-hosted trace and visibility of virtual data

A hypervisor can use TRFCR_EL2.CX to control visibility of CONTEXTIDR_EL2 and VSCTLR_EL2.VMID.

If `SelfHostedTraceEnabled`() == TRUE and TRFCR_EL2.CX == 0:

- The values of CONTEXTIDR_EL2 and VSCTLR_EL2.VMID are not traced.
- Comparisons between CONTEXTIDR_EL2 and VSCTLR_EL2.VMID do not match and results of comparison are not exposed through the comparators.

The trace unit can either prohibit trace for these values, or can record a CONTEXTIDR_EL2 or VSCTLR_EL2.VMID value of zero in the trace.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D3.3 Self-hosted trace timestamps

The trace timestamp is a value that represents the passage of time in real-time. It is calculated from a counter which increments all the time, when the PE is generating trace and when the PE is in a prohibited region.

While SelfHostedTraceEnabled() == FALSE, the external trace provides the trace timestamp. If the external trace is a standard CoreSight system, the relationship between CoreSight time and the Generic Timer counter is IMPLEMENTATION DEFINED.

When SelfHostedTraceEnabled() == TRUE, the trace timestamp is one of the following:

- Physical time, which is defined by the physical count value returned by PhysicalCountInt().
- If FEAT_ECV is implemented, offset physical time, which is defined as the value of (PhysicalCountInt() - CNTPOFF_EL2).
- Virtual time, which is defined as the value of (PhysicalCountInt() - CNTVOFF_EL2). The virtual offset is always CNTVOFF_EL2, including when a read of CNTVCT_EL0 at the current Exception level would treat the virtual offset as zero.

The following fields control which counter is used for self-hosted trace:

- TRFCR_EL2.TS.
- TRFCR_EL1.TS.

The timestamp used for trace is shown in Table D3-2.

In Table D3-2, if CNTHCTL_EL2.ECV is 0 the value of physical offset is zero, otherwise the value of physical offset is the value of CNTPOFF_EL2:

Table D3-2 Timestamp used for trace.

SelfHostedTraceEnabled()	TRFCR_EL2.TS	TRFCR_EL1.TS	Timestamp traced
FALSE	xx	xx	CoreSight time
TRUE	0b00	0b01	PhysicalCountInt() - CNTVOFF_EL2
	0b00	0b10	PhysicalCountInt() - physical offset
	0b00	0b11	PhysicalCountInt()
	0b01	xx	PhysicalCountInt() - CNTVOFF_EL2
	0b10	xx	PhysicalCountInt() - physical offset
	0b11	xx	PhysicalCountInt()

Note

The counter value used for the trace timestamp is not affected by the Effective value of HCR_EL2.E2H, or whether EL2 is enabled or disabled in the current Security state.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D3.4 Synchronization in self-hosted trace

The trace unit is an indirect observer of the System registers.

While SelfHostedTraceEnabled() == TRUE, indirect reads of the trace filter control fields, TRFCR_EL1.{E1TRE, E0TRE} and TRFCR_EL2.{E2TRE, E0HTRE} are treated as indirect reads made by the instruction being traced. For these register fields, in addition to the standard requirements for synchronization of System register accesses, when a trace filter control value is changed and synchronization is not explicitly specified, one of the following occurs:

- The behavior of the PE must be consistent with the control value having the old value.
- The behavior of the PE must change the control value at a point in the simple sequential execution of the program, so that before that point, the behavior of the PE is consistent with the control value having the old value, and after that point the behavior of the PE is consistent with the control value having the new value.

If there are multiple direct writes to the register without explicit synchronization, the behavior is consistent with the writes occurring in program order.

The TSB CSYNC operation is used to ensure that a trace operation, due to a trace unit generating trace for an instruction has completed. The TSB CSYNC operation can be reordered with respect to other instructions, so must be combined with at least one Context synchronization event to ensure the operations are executed in the required order. This means that a direct write to TRFCR_EL1 or TRFCR_EL2 is guaranteed to be observed by the trace unit only after a subsequent Context synchronization event. For more information, see Trace Synchronization Barrier.

While SelfHostedTraceEnabled() == FALSE, the trace unit might impose stronger synchronization requirements.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter D4
The AArch64 System Level Memory Model

This chapter provides a system level view of the general features of the memory system. It contains the following sections:

- About the memory system architecture.
- Address space.
- Mixed-endian support in AArch64.
- Cache support.
- External aborts.
- Memory barrier instructions.
- Pseudocode description of general memory System instructions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D4.1 About the memory system architecture

The Arm architecture supports different implementation choices for the memory system microarchitecture and memory hierarchy, depending on the requirements of the system being implemented. In this respect, the memory system architecture describes a design space in which an implementation is made. The architecture does not prescribe a particular form for the memory systems. Key concepts are abstracted in a way that permits implementation choices to be made while enabling the development of common software routines that do not have to be specific to a particular microarchitectural form of the memory system. For more information about the concept of a hierarchical memory system see Memory hierarchy.

D4.1.1 Form of the memory system architecture

The R-profile architecture includes a *Virtual Memory System Architecture* (VMSA), and a *Protected Memory System Architecture* (PMSA). The AArch64 Virtual Memory System Architecture describes the AArch64 view of the VMSA. The AArch64 Protected Memory System Architecture describes the AArch64 view of the PMSA.

D4.1.2 Memory attributes

Memory types and attributes describes the memory attributes, including how different memory types have different attributes. Each location in memory has a set of memory attributes, and the translation tables define the virtual memory locations, and the attributes for each location.

Table D4-1 shows the memory attributes that are visible at the system level.

Table D4-1 Memory attribute summary

Memory type	Shareability	Cacheability
Device[a]	Outer Shareable	Non-cacheable.
Normal	One of: • Non-shareable. • Inner Shareable. • Outer Shareable.	One of[b]: • Non-cacheable. • Write-Through Cacheable. • Write-Back Cacheable.

a. Takes additional attributes, see Device memory.

b. See also Cacheability, cache allocation hints, and cache transient hints.

For more information on cacheability and shareability see Shareable Normal memory, Non-shareable Normal memory, and Caches and memory hierarchy.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D4.2 Address space

The architecture is designed to support a wide range of applications with different memory requirements. It supports a range of *physical address* (PA) sizes, and provides associated control and identification mechanisms. For more information, see Implemented physical address size.

D4.2.1 Virtual address space overflow

When a PE performs a Simple sequential execution of instructions, it calculates:

```
(address_of_current_instruction) + (size_of_executed_instruction)
```

This calculation is performed after each instruction to determine which instruction to execute next.

If the address calculation performed after executing an instruction overflows `0xFFFF_FFFF_FFFF_FFFF`, the program counter becomes UNKNOWN. Similarly, if the address calculation based on the program counter for the value of the link register or exception link register overflows `0xFFFF_FFFF_FFFF_FFFF`, then those registers also become UNKNOWN.

Note

Address tags are not propagated to the program counter, so the tag does not affect the address calculation.

Where an instruction accesses a sequential set of bytes that crosses the `0xFFFF_FFFF_FFFF_FFFF` boundary when tagged addresses are not used, or the `0xXXFF_FFFF_FFFF_FFFF` boundary when tagged addresses are used, then the virtual address accessed for the bytes above this boundary is UNKNOWN.

When tagged addresses are used, the value of the tag associated with the address also becomes UNKNOWN.

Note

The behaviors described in this section apply only for the upper bound of the upper VA range, in translation regimes that have two VA ranges. They do not apply for address calculations relating to the top of the lower VA range.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D4.3 Mixed-endian support in AArch64

A control bit, SCTLR_EL1.E0E is provided to allow the endianness of explicit data accesses made while executing at EL0 to be controlled independently of those made while executing at EL1. Table D4-2 shows the endianness of explicit data accesses and translation table walks.

Table D4-2 Endianness support

Exception level	Explicit data accesses	Stage 1 translation table walks
EL0	SCTLR_EL1.E0E	SCTLR_EL1.EE
EL1	SCTLR_EL1.EE	SCTLR_EL1.EE
EL2	SCTLR_EL2.EE	SCTLR_EL2.EE

Note

SCTLR_EL1.E0E has no effect on the endianness of the Explicit Memory Effects generated by unprivileged memory access instructions, when these are executed at EL1.

AArch64 state provides the following options for endianness support:

- All Exception levels support mixed-endianness:
  - SCTLR_ELx.EE, SCTLR_EL1.E0E, and SCTLR_EL2.E0E are RW.
- Only EL0 supports mixed-endianness and EL1and EL2 support only little-endianness:
  - SCTLR_ELx.EE is RES0, and SCTLR_EL1.E0E and SCTLR_EL2.E0E are RW.
- Only EL0 supports mixed-endianness and EL1and EL2 support only big-endianness:
  - SCTLR_ELx.EE is RES1, and SCTLR_EL1.E0E and SCTLR_EL2.E0E are RW.
- All Exception levels support only little-endianness:
  - SCTLR_ELx.EE, SCTLR_EL1.E0E, and SCTLR_EL2.E0E are RES0.
- All Exception levels support only big-endianness:
  - SCTLR_ELx.EE, SCTLR_EL1.E0E, and SCTLR_EL2.E0E are RES1.

The `BigEndian()` function determines whether the current Exception level and Execution state are using big-endian data. This function is defined in A-profile Architecture Pseudocode.

For more information about endianness in the Arm architecture see Endian support.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D4.4 Cache support

This section describes the cache identification and control mechanisms, and the A64 cache maintenance instructions, in the following sections:

- General behavior of the caches.
- Cache identification.
- Cacheability, cache allocation hints, and cache transient hints.
- Enabling and disabling the caching of memory accesses.
- Behavior of caches at reset
- Non-cacheable accesses and instruction caches.
- About cache maintenance in AArch64 state.
- A64 Cache maintenance instructions
- Data cache zero instruction.
- Cache lockdown.
- System level caches.
- Branch prediction.
- Execution, data prediction and prefetching restriction System instructions.

See also Caches.

D4.4.1 General behavior of the caches

When a memory location has a Normal Cacheable memory attribute, determining whether a copy of the memory location is held in a cache still depends on many aspects of the implementation. The following non-exhaustive list of factors might be involved:

- The size, line length, and associativity of the cache.
- The cache allocation algorithm.
- Activity by other elements of the system that can access the memory.
- Speculative instruction fetching algorithms.
- Speculative data fetching algorithms.
- Interrupt behaviors.

Given this range of factors, and the large variety of cache systems that might be implemented, the architecture cannot guarantee whether:

- A memory location present in the cache remains in the cache.
- A memory location not present in the cache is brought into the cache.

Instead, the following principles apply to the behavior of caches:

- The architecture has a concept of an entry locked down in the cache. How lockdown is achieved is IMPLEMENTATION DEFINED, and lockdown might not be supported by:
  - A particular implementation.
  - Some memory attributes.
- An unlocked entry in a cache might not remain in that cache. The architecture does not guarantee that an unlocked cache entry remains in the cache or remains incoherent with the rest of memory. Software must not assume that an unlocked item that remains in the cache remains dirty.
- A locked entry in a cache is guaranteed to remain in that cache. The architecture does not guarantee that a locked cache entry remains incoherent with the rest of memory, that is, it might not remain dirty.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

For more information, see The interaction of cache lockdown with cache maintenance instructions.

- Any memory location that has a Normal Cacheable attribute at either the current Exception level or at a higher Exception level can be allocated to a cache at any time.
- It is guaranteed that no memory location will be allocated into a Data or Unified cache if that location does not have a Normal Cacheable attribute in either:
  - The translation regime at the current Exception level.
  - The translation regime at any higher Exception level.
- For data accesses, any memory location with a Normal Inner Shareable or Normal Outer Shareable attribute is guaranteed to be coherent with all Requesters in its shareability domain.
- Any memory location is not guaranteed to remain incoherent with the rest of memory.
- The eviction of a cache entry from a cache level can overwrite memory that has been written by another observer only if the entry contains a memory location that has been written to by an observer in the shareability domain of that memory location. The maximum size of the memory that can be overwritten is called the *Cache Write-back Granule*. In some implementations the CTR_EL0 identifies the Cache Write-back Granule.
- The allocation of a memory location into a cache cannot cause the most recent value of that memory location to become invisible to an observer if it was previously visible to that observer.

Note

The Cacheability attribute of an address is determined by the applicable translation table entry for that address, as modified by any applicable System register Cacheability controls, such as the SCTLR_EL1.{I, C} controls.

For the purpose of these principles, a cache entry covers at least 16 bytes and no more than 2KB of contiguous address space, aligned to the size of the cache entry.

D4.4.2 Cache identification

The cache identification registers describe the implemented caches that are affected by cache maintenance instructions executed on the PE. This includes the cache maintenance instructions that:

- Affect the entire cache, for example IC IALLU.
- Operate by VA, for example IC IVAU.
- Operate by set/way, for example DC ISW.

The cache identification registers are:

- The Cache Type Register, CTR_EL0, that defines:
  - The minimum line length of any of the instruction caches affected by the instruction cache maintenance instructions.
  - The minimum line length of any of the data or unified caches, affected by the data cache maintenance instruction.
  - The cache indexing and tagging policy of the Level 1 instruction cache.

  Note

  It is IMPLEMENTATION DEFINED whether caches beyond the PoC will be reported by this mechanism, and because of the possible existence of system caches some caches before the PoC might not be reported. For more information about system caches see System level caches.
- A single Cache Level ID Register, CLIDR_EL1, that defines:
  - The type of cache that is implemented and can be maintained using the architected cache maintenance instructions that operate by set/way or operate on the entire cache at each cache level, up to the maximum of seven levels.
  - The *Level of Coherence* (LoC) for the caches. See Terms used in describing the cache maintenance instructions for the definition of LoC.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - The *Level of Unification Uniprocessor* (LoUU) for the caches. See Terms used in describing the cache maintenance instructions for the definition of LoUU.
  - An optional ICB field to indicate the boundary between the caches use for caching Inner Cacheable memory regions and those used only for caching Outer Cacheable regions.
- A single Cache Size Selection Register, CSSELR_EL1, that selects the cache level and sort of cache (Instruction, Data/Unified/Tag) of the current Cache Size Identification Register.
- For each implemented cache that is identifiable by this mechanism, across all the levels of caching, a Cache Size Identification Register, CCSIDR_EL1, that defines:
  - The number of sets, associativity and line length of the cache. See Terms used in describing the cache maintenance instructions for a definition of these terms.

Note

From Armv8.3, multiple formats of the Cache Size Identification Register are supported. For more information, see Possible formats of the Cache Size Identification Register, CCSIDR_EL1.

To determine the cache topology associated with a PE:

1. Read the Cache Type Register to find the indexing and tagging policy used for the Level 1 instruction cache. This register also provides the size of the smallest cache lines used for the instruction caches, and for the data and unified caches. These values are used in cache maintenance instructions.
2. Read the Cache Level ID Register to find what caches are implemented. The register includes seven Cache type fields, for cache levels 1 to 7. Scanning these fields, starting from Level 1, identifies the instruction, data or unified caches implemented at each level. This scan ends when it reaches a level at which no caches are defined. The Cache Level ID Register also specifies the Level of Unification (LoU) and the Level of Coherence (LoC) for the cache implementation.
3. For each cache identified at stage 2:
   - Write to the Cache Size Selection Register to select the required cache. A cache is identified by its level, and whether it is:
     - An instruction cache.
     - A data or unified cache.
   - Read the Cache Size Identification Register to find details of the cache.

D4.4.2.1 Possible formats of the Cache Size Identification Register, CCSIDR_EL1

From Armv8.3, the Cache Size Identification Register, CCSIDR_EL1 has two different formats available for defining the number of sets and associativity of the cache. For a definition of these terms, see Terms used in describing the cache maintenance instructions.

When FEAT_CCIDX is implemented:

- CCSIDR_EL1 is a 64-bit register.
- The length of the CCSIDR_EL1.Assoc field is 21 bits. This limits the associativity of the currently selected cache to 2^{21}.
- The length of the CCSIDR_EL1.NumSets field is 24 bits. This limits the number of sets in the currently selected cache to 2^{24}.

This is the 64-bit format of the Cache Size Identification Register.

When FEAT_CCIDX is not implemented:

- CCSIDR_EL1 is a 32-bit register.
- The length of the CCSIDR_EL1.Assoc field is 10 bits. This limits the associativity of the currently selected cache to 2^{10}.
- The length of the CCSIDR_EL1.NumSets field is 15 bits. This limits the number of sets in the currently selected cache to 2^{15}.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This is the 32-bit format of the Cache Size Identification Register.

When one of these formats is implemented, it is implemented across all the levels of caching.

D4.4.3 Cacheability, cache allocation hints, and cache transient hints

Cacheability applies only to Normal memory, and can be defined independently for Inner and Outer cache locations. All types of Device memory are always treated as Non-cacheable.

As described in Memory types and attributes, the memory attributes include a cacheability attribute that is one of:

- Non-cacheable.
- Write-Through cacheable.
- Write-Back cacheable.

Cacheability attributes other than Non-cacheable can be complemented by a *cache allocation hint*. This is an indication to the memory system of whether allocating a value to a cache is likely to improve performance. In addition, it is IMPLEMENTATION DEFINED whether a *cache transient hint* is supported, see Transient cacheability hint.

The cache allocation hints are assigned independently for read and write accesses, and therefore when the Transient hint is supported the following cache allocation hints can be assigned:

For read accesses: Read-Allocate, Transient Read-Allocate, or No Read-Allocate.

For write accesses: Write-Allocate, Transient Write-Allocate, or No Write-Allocate.

Note

- A Cacheable location with both No Read-Allocate and No Write-Allocate hints is not the same as a Non-cacheable location. A Non-cacheable location has coherency guarantees for all observers within the system that do not apply for a location that is Cacheable, No Read-Allocate, No Write-Allocate.
- Implementations can use the cache allocation hints to limit cache pollution to a part of a cache, such as to a subset of ways.
- For VMSAv8-64 translation table walks, the TCR_EL1.{IRGN*n*, ORGN*n*} fields define the memory attributes of the translation tables, including the cacheability. However, this assignment supports only a subset of the cacheability attributes described in this section.

The architecture does not require an implementation to make any use of cache allocation hints. This means an implementation might not make any distinction between memory locations with attributes that differ only in their cache allocation hint.

D4.4.3.1 Transient cacheability hint

In Armv8, it is IMPLEMENTATION DEFINED whether a Transient hint is supported. In an implementation that supports the Transient hint, the Transient hint is a qualifier of the cache allocation hints, and indicates that the benefit of caching is for a relatively short period. It indicates that it might be better to restrict allocation of transient entries, to avoid possibly casting-out other, less transient, entries.

Note

The architecture does not specify what is meant by a *relatively short period*.

The description of the AArch64 MAIR_EL1, MAIR_EL2 registers, includes the assignment of the Transient hint in an implementation that supports this option. In this assignment:

- The Transient hint is defined independently for Inner Cacheable and Outer Cacheable memory regions.
- A single Transient hint applies to both read and write accesses to a memory region.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D4.4.4 Enabling and disabling the caching of memory accesses

Cacheability control fields can force all memory locations with the Normal memory type to be treated as Non-cacheable, regardless of their assigned Cacheability attribute. Independent controls are provided for each stage of address translation, with separate controls for:

- Data accesses. These controls also apply to accesses to the translation tables.
- Instruction accesses.

Note

These Cacheability controls replace the cache enable controls provided in previous versions of the Arm architecture.

The Cacheability control fields and their effects are as follows:

For the EL1&0 translation regime

- When the value of SCTLR_EL1.C is 0:
  - All stage 1 translations for data accesses to Normal memory are Non-cacheable.
  - All accesses to the EL1&0 stage 1 translation tables are Non-cacheable.
- When the value of SCTLR_EL1.I is 0:
  - All stage 1 translations for instruction accesses to Normal memory are Non-cacheable.
- When the value of HCR_EL2.CD is 1 all stage 2 translations for data accesses to Normal memory are Non-cacheable.
- When the value of HCR_EL2.ID is 1 all stage 2 translations for instruction accesses to Normal memory are Non-cacheable.
- When the value of HCR_EL2.DC is 1, all stage 1 translations and all accesses to the EL1&0 stage 1 translation tables, are treated as accesses to Normal Non-shareable Inner Write-Back Cacheable Read-Allocate Write-Allocate, Outer Write-Back Cacheable Read-Allocate Write-Allocate memory, regardless of the value of SCTLR_EL1.{I, C}. This applies to translations for both data and instruction accesses.
- The stage 1 and stage 2 cacheability attributes are combined as described in Combining stage 1 and stage 2 Cacheability attributes for Normal memory.
- The SCTLR_EL1.{C, I} and HCR_EL2.DC fields have no effect on the EL2 translation regimes.
- The HCR_EL2.{ID, CD} fields affect only stage 2 of the EL1&0 translation regime.

D4.4.5 Behavior of caches at reset

The behavior of caches at reset is as follows:

- All caches reset to IMPLEMENTATION DEFINED states that might be UNKNOWN.
- The Cacheability control fields described in Enabling and disabling the caching of memory accesses reset to values that force all memory locations to be treated as Non-cacheable.

  Note

  This applies only to the controls that apply to the Translation regime that is used by the Exception level and Security state entered on reset.

- An implementation can require the use of a specific cache initialization routine to invalidate its storage array before caching is enabled. The exact form of any required initialization routine is IMPLEMENTATION DEFINED, and the routine must be documented clearly as part of the documentation of the device.
- If an implementation permits cache hits when the Cacheability control fields force all memory locations to be treated as Non-cacheable then the cache initialization routine must:
  - Provide a mechanism to ensure the correct initialization of the caches.
  - Be documented clearly as part of the documentation of the device.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

In particular, if an implementation permits cache hits when the Cacheability controls force all memory locations to be treated as Non-cacheable, and the cache contents are not invalidated at reset, the initialization routine must avoid any possibility of running from an uninitialized cache. It is acceptable for an initialization routine to require a fixed instruction sequence to be placed in a restricted range of memory.

- Arm recommends that whenever an invalidation routine is required, it is based on the cache maintenance instructions.

See also TLB behavior at reset.

D4.4.6 Non-cacheable accesses and instruction caches

In AArch64 state, instruction accesses to Non-cacheable Normal memory can be held in instruction caches.

Correspondingly, the sequence for ensuring that modifications to instructions are available for execution must include invalidation of the modified locations from the instruction cache, even if the instructions are held in Normal Non-cacheable memory. This includes cases where System register Cacheability control fields force instruction accesses to memory to be Non-cacheable.

Therefore when using self-modified code in Non-cacheable space in a uniprocessor system, the following sequence is required:

```
; Enter this code with <Wt> containing the new 32-bit instruction
; to be held at a location pointed to by <Xn> in Normal Non-cacheable memory.
STR <Wt>, [Xn]
DSB ISH; Ensure visibility of the data stored
IC IVAU, [Xn] ; Invalidate instruction cache by VA to PoU
DSB ISH; Ensure completion of the invalidations
ISB ;
```

In a multiprocessor system, the IC IVAU for a non-cacheable location is broadcast to all PEs within the Inner Shareable domain of the PE running this sequence. This is despite non-cacheable normal memory locations being treated as Outer Shared in other parts of the architecture.

Additional software steps might be required to synchronize the threads with other PEs. This might be necessary so that the PEs executing the modified instructions can execute an `ISB` after completing the invalidation, and to avoid issues associated with concurrent modification and execution of instruction sequences. See also Concurrent modification and execution of instructions and Concurrent modification and execution of instructions.

Larger blocks of instructions can be modified using the IC IALLU instruction for a uniprocessor system, or an IC IALLUIS for a multiprocessor system.

Note

This section applies even when the Cacheability control fields force instruction accesses to memory in AArch64 state to be Non-cacheable, as described in Enabling and disabling the caching of memory accesses.

D4.4.7 About cache maintenance in AArch64 state

The following sections give general information about cache maintenance:

- Terms used in describing the cache maintenance instructions.
- Abstraction of the cache hierarchy.

The following sections describe the A64 cache maintenance instructions:

- The instruction cache maintenance instruction (IC).
- The data cache maintenance instruction (DC).

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Some descriptions of the cache maintenance instructions refer to the cacheability of the address on which the instruction operates. The Cacheability of an address is determined by the applicable translation table entry for that address, as modified by any applicable System register Cacheability controls, such as the SCTLR_EL1.{I, C} controls.

D4.4.7.1 Terms used in describing the cache maintenance instructions

A *memory location* is a byte that is associated with an address in a particular address space. For example, address `0x1000` in the Physical address space is a different memory location from address `0x1000` in the Secure physical address space.

The term *Resource* means a physical entity that can be accessed at one or more memory locations. A Resource *associated with* a physical address space is accessible in that physical address space.

Note

An example of a Resource is a byte of memory that can be accessible at a fixed address but in different physical address spaces, determined by a configuration option.

Cache maintenance instructions are defined to act on particular memory locations. Depending on the instruction type, the scope is defined as one of:

- By the virtual address of the memory location to be maintained, referred to as operating *by VA*.
- By the physical address of the memory location to be maintained, referred to as operating *by PA*.
- By a mechanism that describes the location in the hardware of the cache, referred to as operating *by set/way*.

In addition, for instruction caches, there are instructions that invalidate all entries.

The following subsections define the terms used in the descriptions of the cache maintenance instructions:

- Terminology for cache maintenance instructions operating by set/way.
- Terminology for Clean, Invalidate, and Clean and Invalidate instructions.

Note

There is no terminology specific to cache maintenance instructions that operate by VA. When all applicable stages of translation are disabled, the VA used is identical to the PA. For more information about memory system behavior when address translation is disabled, see The effects of disabling an address translation stage.

D4.4.7.1.1 Terminology for cache maintenance instructions operating by set/way

Cache maintenance instruction that operate by set/way refer to the particular structures in a cache. Three parameters describe the location in a cache hierarchy that an instruction works on. These parameters are:

Level The cache level of the hierarchy. The number of levels of cache is IMPLEMENTATION DEFINED. The cache levels that can be managed using the architected cache maintenance instructions that operate by set/way can be determined from the CLIDR_EL1.

In the Arm architecture, the lower numbered cache levels are those closest to the PE. See Memory hierarchy.

Set Each level of a cache is split up into a number of *sets*. Each set is a set of locations in a cache level to which an address can be assigned. Usually, the set number is an IMPLEMENTATION DEFINED function of an address.

In the Arm architecture, sets are numbered from 0.

Way The associativity of a cache is the number of locations in a set to which a specific address can be assigned. The *way* number specifies one of these locations.

In the Arm architecture, ways are numbered from 0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Because the allocation of a memory address to a cache location is entirely IMPLEMENTATION DEFINED, Arm expects that most portable software will use only the cache maintenance instructions by set/way as single steps in a routine to perform maintenance on the entire cache.

D4.4.7.1.2 Terminology for Clean, Invalidate, and Clean and Invalidate instructions

Caches introduce coherency problems in two possible directions:

1. An update to a memory location by a PE that accesses a cache might not be visible to other observers that can access memory. This can occur because new updates are still in the cache and are not visible yet to the other observers that do not access that cache.
2. Updates to memory locations by other observers that can access memory might not be visible to a PE that accesses a cache. This can occur when the cache contains an old, or *stale*, copy of the memory location that has been updated.

The *Clean* and *Invalidate* instructions address these two issues. The definitions of these instructions are:

Clean A cache clean instruction ensures that updates made by an observer that controls the cache are made visible to other observers that can access memory at the point to which the instruction is performed. Once the Clean has completed, the new memory values are guaranteed to be visible to the point to which the instruction is performed, for example to the Point of Unification.

The cleaning of a cache entry from a cache can overwrite memory that has been written by another observer only if the entry contains a location that has been written to by an observer in the shareability domain of that memory location.

Invalidate A cache invalidate instruction ensures that updates made visible by observers that access memory at the point to which the invalidate is defined, are made visible to an observer that controls the cache. This might result in the loss of updates to the locations affected by the invalidate instruction that have been written by observers that access the cache, if those updates have not been cleaned from the cache since they were made.

If the address of an entry on which the invalidate instruction operates is Normal, Non-cacheable or any type of Device memory then an invalidate instruction also ensures that this address is not present in the cache.

Note

Entries for addresses that are Normal Cacheable can be allocated to the cache at any time, and so the cache invalidate instruction cannot ensure that the address is not present in a cache.

Clean and Invalidate

A cache *clean and invalidate* instruction behaves as the execution of a clean instruction followed immediately by an invalidate instruction. Both instructions are performed to the same location.

The points to which a cache maintenance instruction can be defined differ depending on whether the instruction operates by VA or by set/way:

- For instructions operating by set/way, the point is defined to be to the next level of caching. For the All operations, the point is defined as the Point of Unification for each location held in the cache.
- For instructions operating by VA, the following conceptual points are defined:

Point of Coherency (PoC)

The point at which all agents that can access memory are guaranteed to see the same copy of a memory location for accesses of any memory type or cacheability attribute. In many cases this is effectively the main system memory, although the architecture does not prohibit the implementation of caches beyond the PoC that have no effect on the coherency between memory system agents.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

The presence of system caches can affect the determination of the Point of Coherency as described in System level caches.

Point of Unification (PoU)

The PoU for a PE is the point by which the instruction and data caches and the translation table walks of that PE are guaranteed to see the same copy of a memory location. In many cases, the Point of Unification is the point in a uniprocessor memory system by which the instruction and data caches and the translation table walks have merged.

The PoU for an Inner Shareable shareability domain is the point by which the instruction and data caches and the translation table walks of all the PEs in that Inner Shareable shareability domain are guaranteed to see the same copy of a memory location. Defining this point permits self-modifying software to ensure future instruction fetches are associated with the modified version of the software by using the standard correctness policy of:

1. Clean data cache entry by address.
2. Invalidate instruction cache entry by address.

Point of Persistence (PoP)

When FEAT_DPB is implemented: The point in a memory system, if it exists, at or beyond the Point of Coherency, where a write to memory is maintained when system power is removed, and reliably recovered when power is restored to the affected locations in memory.

When FEAT_DPB and FEAT_DPB2 are implemented: The point in a memory system where there is a system guarantee that there is sufficient energy within the system to ensure that a write to memory will be persistent if system power is removed.

Note

Such memory is sometimes called non-volatile memory. For example, the Storage-class memory shown in Figure B1-3 could be used as target memory for this feature.

Point of Deep Persistence (PoDP) The point in a memory system where any writes that have reached that point are persistent, even in the event of an instantaneous hardware failure of the power system.

The following fields in the CLIDR_EL1 relate to the PoC and PoU:

LoC, Level of Coherence

This field defines the last level of cache that must be cleaned or invalidated when cleaning or invalidating to the Point of Coherency. The LoC value is a cache level, so, for example, if LoC contains the value 3:

- A clean to the Point of Coherency operation requires the level 1, level 2 and level 3 caches to be cleaned.
- Level 4 cache is the first level that does not have to be maintained.

If the LoC field value is `0x0`, this means that no levels of cache need to cleaned or invalidated when cleaning or invalidating to the Point of Coherency.

If the LoC field value is a nonzero value that corresponds to a level that is not implemented, this indicates that all implemented caches are before the Point of Coherency.

LoUU, Level of Unification, uniprocessor

This field defines the last level of data cache that must be cleaned, or the last level of instruction cache that must be invalidated, when cleaning or invalidating to the Point of Unification for the PE. As with LoC, the LoUU value is a cache level.

If the LoUU field value is `0x0`, this means that no levels of data cache need to be cleaned or invalidated when cleaning or invalidating to the Point of Unification.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If the LoUU field value is a nonzero value that corresponds to a level that is not implemented, this indicates that all implemented caches are before the Point of Unification.

LoUIS, Level of Unification, Inner Shareable

In any implementation:

- This field defines the last level of data or unified cache that must be cleaned, or the last level of instruction or unified cache that must be invalidated, when cleaning or invalidating to the Point of Unification for the Inner Shareable shareability domain. As with LoC, the LoUIS value is a cache level.
- If the LoUIS field value is `0x0`, this means that no levels of data or unified cache need to cleaned or invalidated when cleaning or invalidating to the Point of Unification for the Inner Shareable shareability domain.
- If the LoUIS field value is a nonzero value that corresponds to a level that is not implemented, this indicates that all implemented caches are before the Point of Unification.

D4.4.7.2 Abstraction of the cache hierarchy

The following subsections describe the abstraction of the cache hierarchy:

- Cache maintenance instructions that operate by VA.
- Cache maintenance instructions that operate by set/way.

D4.4.7.2.1 Cache maintenance instructions that operate by VA

The VA-based cache maintenance instructions are described as operating by VA. Each of these instructions is always qualified as being one of:

- Performed to the Point of Coherency.
- Performed to the Point of Unification.
- When FEAT_DPB is implemented, performed to the Point of Persistence.
- When FEAT_PoPS is implemented, performed to the Point of Physical Storage.

See Terms used in describing the cache maintenance instructions for definitions of these terms, and for more information about possible meanings of VA.

A64 Cache maintenance instructions lists the VA-based maintenance instructions.

The CTR_EL0 holds minimum line length values for:

- The instruction caches.
- The data and unified caches.

These values support efficient invalidation of a range of VAs, because this value is the most efficient address stride to use to apply a sequence of VA-based maintenance instructions to a range of VAs.

For the Invalidate data or unified cache line by VA instruction, the Cache Write-back Granule field of the CTR_EL0 defines the maximum granule that a single invalidate instruction can invalidate. This meaning of the Cache Write-back Granule is in addition to its defining the maximum size that can be written back.

D4.4.7.2.2 Cache maintenance instructions that operate by set/way

A64 Cache maintenance instructions lists the set/way-based maintenance instructions. Some encodings of these instructions include a required field that specifies the cache level for the instruction:

- A clean instruction cleans from the level of cache specified through to at least the next level of cache, moving further from the PE.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- An invalidate instruction invalidates only at the level specified.

D4.4.8 A64 Cache maintenance instructions

The A64 cache maintenance instructions are part of the A64 System instruction class in the register encoding space. For encoding details and other general information on these System instructions, see System instructions, SYS and Cache maintenance instructions, and data cache zero operations.

The following subsections give more information about these instructions:

- The instruction cache maintenance instruction (IC).
- The data cache maintenance instruction (DC).
- EL0 accessibility of cache maintenance instructions.
- General requirements for the scope of maintenance instructions.
- Effects of instructions that operate by VA to the PoC.
- Effects of instructions that operate by VA to the PoP.
- Effects of instructions that operate by VA to the PoPS.
- Effects of instructions that operate by VA to the PoU.
- Effects of All and set/way maintenance instructions.
- Effects of virtualization and Security state on the cache maintenance instructions.
- Boundary conditions for cache maintenance instructions.
- Ordering and completion of data and instruction cache instructions.
- Performing cache maintenance instructions.

D4.4.8.1 The instruction cache maintenance instruction (`IC`)

System instructions describes the A64 assembly syntax for this instruction.

When an `IC` instruction requires an address argument this takes the form of a 64-bit register that holds the VA argument. No alignment restrictions apply for this address.

Any cache maintenance instruction operating by VA includes as part of any required VA to PA translation:

- For an instruction executed at EL1, or at EL2 when the Effective value of HCR_EL2.E2H is 1, the current ASID.
- The current Security state.
- Whether the instruction was executed at EL1 or EL2.
- For an instruction executed at EL1, the current VMID.

That VA to PA translation might fault. However, for an instruction cache maintenance instruction that operates by VA:

- It is IMPLEMENTATION DEFINED whether the instruction can generate:
  - An Access flag fault.
  - A Translation fault.
- The instruction cannot generate a Permission fault, except for the possible generation of a Permission fault by the execution of an IC IVAU instruction at EL0 when the specified address does not have read access at EL0, as described in EL0 accessibility of cache maintenance instructions.

For more information about possible faults on a cache maintenance instruction that operates by VA, see Memory aborts.

See also Ordering and completion of data and instruction cache instructions.

If CTR_EL0.DIC is 1, then it is IMPLEMENTATION DEFINED whether each of the IC instructions is treated as a `NOP`.

If an IC instruction that operates by VA is treated as a `NOP`, it does not perform translation and all the following apply:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- No MMU faults can be generated.
- If hardware update of AF is enabled, AF is not required to be set.

If an IC instruction is treated as a NOP, then it is IMPLEMENTATION DEFINED whether the execution of the instruction can be trapped.

D4.4.8.2 The data cache maintenance instruction (DC)

System instructions describes the A64 assembly syntax for this instruction.

When a DC instruction requires a set/way/level argument this takes the form of a 64-bit register, the upper 32 bits of which are RES0.

If a data cache maintenance by set/way instruction specifies a set, way, or level argument that is larger than the value supported by the implementation then the instruction is CONSTRAINED UNPREDICTABLE, see Out of range values of the Set/Way/Index fields in cache maintenance instructions or the instruction description.

When a DC instruction requires an address argument this takes the form of a 64-bit register that holds the VA argument. No alignment restrictions apply for this address.

Any cache maintenance instruction operating by VA includes as part of any required VA to PA translation:

- For an instruction executed at EL1, or at EL2 when the Effective value of HCR_EL2.E2H is 1, the current ASID.
- The current Security state.
- Whether the instruction is executed at EL1 or EL2.
- For an instruction executed at EL1, the current VMID.

That VA to PA translation might fault. However, a data or unified cache maintenance instruction that operates by VA cannot generate a Permission fault except in the following cases:

- The possible generation of a Permission fault by:
  - The execution of a DC IVAC instruction when the specified address does not have write permission.
  - The execution of an enabled DC instruction at EL0 when the specified address does not have read access at EL0, as described in EL0 accessibility of cache maintenance instructions.

  The description of Permission faults includes possible constraints on the generation of Permission faults on cache maintenance by VA instructions.

For more information about possible faults on a VA to PA translation, see Memory aborts.

When executed at EL1, a DC ISW instruction performs a clean and invalidate, meaning it performs the same maintenance as a DC CISW instruction, if all of the following apply:

- EL2 is implemented and enabled in the current Security state.
- Either:
  - The value of HCR_EL2.SWIO is 1, forcing a cache clean to perform a clean and invalidate.
  - The value of HCR_EL2.VM is 1, meaning EL1&0 stage two address translation is enabled.

When executed at EL1, a DC IVAC instruction performs a clean and invalidate, meaning it performs the same maintenance as a DC CIVAC instruction, if all of the following apply:

- EL2 is implemented and enabled in the current Security state.
- The value of HCR_EL2.VM is 1, meaning EL1&0 stage two address translation is enabled.

When FEAT_DPB is implemented, meaning the DC CVAP instruction is implemented, if the memory system does not support the Point of Persistence, a data cache clean to the PoP, DC CVAP, behaves as a data cache clean to the PoC, DC CVAC.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

- Support for the Point of Persistence does not change the definition or behavior of the CLIDR_EL1 System register.
- Because a `DSB SYS` instruction will not complete until all previous DC CVAP instructions have completed, the following sequence can be used to ensure the completion of any store to the Point of Persistence, where the store might be to Non-cacheable memory:

```
DMB            ; Note this can be any DMB that applies to both loads and stores
DC CVAP, Xt
DSB SYS
```

- If caches that are invisible to the programmer exist beyond the Point of Coherency but before the Point of Persistence and hold data that is marked as Non-cacheable, the DC CVAP operation causes the Non-cacheable locations to be cleaned from those caches.

If there are caches after the Point of Coherency and FEAT_PoPS is not implemented, then the DC CIVAC and DC CIGDVAC instructions are not sufficient to remove all copies of a poisoned Location and it is IMPLEMENTATION DEFINED whether any IMPLEMENTATION DEFINED mechanism exists to remove poison from a Location.

See Clearing Deferred errors from poisoned Locations for more information on removing poison from a Location.

If a memory fault that sets the FAR for the translation regime applicable for the cache maintenance instruction is generated from a data cache maintenance instruction, the FAR holds the address specified in the register argument of the instruction.

See also EL0 accessibility of cache maintenance instructions and Ordering and completion of data and instruction cache instructions.

For all the following DC instructions, it is IMPLEMENTATION DEFINED whether the instruction is treated as a `NOP`:

- A DC to the PoU and the LoUIS and LoUU are 0.
- A DC to the PoC and the LoC is 0.
- A DC to the PoP and the memory system does not support the Point of Persistence, and the LoC is 0.
- A DC to the PoDP and the memory system does not support the Point of Deep Persistence, and does not support the Point of Persistence, and the LoC is 0.

If a DC instruction that operates by VA is treated as a `NOP`, it does not perform translation and all the following apply:

- No MMU or MPU faults can be generated.
- If hardware update of AF is enabled, AF is not required to be set.

If a DC instruction is treated as a `NOP`, then it is IMPLEMENTATION DEFINED whether the execution of the instruction can be trapped.

D4.4.8.3 EL0 accessibility of cache maintenance instructions

Software executing at EL0 can access data cache maintenance instructions and instruction cache maintenance instructions that operate by virtual address. When EL0 use of these instructions is disabled because SCTLR_EL1.UCI is 0, executing one of these instructions at EL0 generates a trap to EL1, which is reported using EC = `0x18`.

Note

DC CVAP is implemented only if FEAT_DPB is implemented.

For these instructions read access permission for the virtual address is required. When the value of SCTLR_EL1.UCI is 1:

- If the `DC` instruction is executed at EL0 and the address specified in the argument cannot be read at EL0, a Permission fault might be generated.
- For the IC IVAU instruction, if the instruction is executed at EL0 and the address specified in the argument cannot be read at EL0, it is IMPLEMENTATION DEFINED whether a Permission fault is generated.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For more information, see the description of Permission faults, In the case of a DC instruction executed at EL0 when the address specified cannot be read at EL0 the Permission fault is generated unless one of the permitted constraints described in that section applies and means the fault cannot be generated.

Software can read the CTR_EL0 to discover the stride needed for cache maintenance instructions. The SCTLR_EL1.UCT bit enables EL0 access to the CTR_EL0. When EL0 access to the Cache Type register is disabled, a register access instruction executed at EL0 is trapped to EL1 using EC = 0x18.

D4.4.8.4 General requirements for the scope of maintenance instructions

The specification of the cache maintenance instructions describes what each instruction is guaranteed to do in a system. It does not limit other behaviors that might occur, provided they are consistent with the requirements described in General behavior of the caches, Behavior of caches at reset, and Prefetching into cache.

This means that as a side-effect of a cache maintenance instruction:

- Any location in the cache might be cleaned.
- Any unlocked location in the cache might be cleaned and invalidated.

Note

Arm recommends that, for best performance, such side-effects are kept to a minimum. Arm strongly recommends that the side-effects of operations performed in one Security state do not have a significant performance impact on execution in another Security state.

If an implementation can overwrite Allocation Tags in memory that have been written by another observer, where the Allocation Tags have not been written by an observer in the Shareability domain of that memory location, then:

- A cache maintenance operation which cleans data from a cache level must also clean the associated Allocation Tags.
- A cache maintenance operation which invalidates, or cleans and invalidates data from a cache level, must also clean and invalidate the associated Allocation Tags.

D4.4.8.5 Effects of instructions that operate by VA to the PoC

For Normal memory that is not Inner Non-cacheable, Outer Non-cacheable, cache maintenance instructions that operate by VA to the PoC affect the caches of other PEs in the shareability domain described by the shareability attributes of the VA supplied with the instruction.

For Device memory or Normal memory that is Inner Non-cacheable, Outer Non-cacheable, these instructions affect the caches of all PEs in the Outer Shareable shareability domain of the PE on which the instructions are operating.

In all cases, for any affected PE, these instructions affect all data and unified caches to the PoC. Table D4-3 shows the scope of these Data and unified cache maintenance instructions.

Table D4-3 PEs affected by cache maintenance instructions to the PoC

Shareability	PEs affected	Effective to
Non-shareable	The PE executing the instruction	The PoC of the entire system
Inner Shareable	All PEs in the same Inner Shareable shareability domain as the PE executing the instruction	The PoC of the entire system
Outer Shareable	All PEs in the same Outer Shareable shareability domain as the PE executing the instruction	The PoC of the entire system

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

It is IMPLEMENTATION DEFINED by the system whether the cache maintenance instructions have an effect on the caches of observers that are not PEs within the affected shareability domain to which the cache maintenance instructions apply.

D4.4.8.6 Effects of instructions that operate by VA to the PoP

For Normal memory that is not Inner Non-cacheable, Outer Non-cacheable, cache maintenance instructions that operate by VA to the PoP affect the caches of other PEs in the shareability domain described by the shareability attributes of the VA supplied with the instruction.

For Device memory or Normal memory that is Inner Non-cacheable, Outer Non-cacheable, these instructions affect the caches of all PEs in the Outer Shareable shareability domain of the PE on which the instructions are operating.

In all cases, for any affected PE, these instructions affect all data and unified caches to the PoP. Table D4-4 shows the scope of these Data and unified cache maintenance to the PoP instructions.

Table D4-4 PEs affected by cache maintenance instructions to the PoP

Shareability	PEs affected	Effective to
Non-shareable	The PE executing the instruction	The PoP of the entire system
Inner Shareable	All PEs in the same Inner Shareable shareability domain as the PE executing the instruction	The PoP of the entire system
Outer Shareable	All PEs in the same Outer Shareable shareability domain as the PE executing the instruction	The PoP of the entire system

Note

It is IMPLEMENTATION DEFINED by the system whether the cache maintenance instructions have an effect on the caches of observers that are not PEs within the affected shareability domain to which the cache maintenance instructions apply.

D4.4.8.7 Effects of instructions that operate by VA to the PoPS

For Normal memory that is not Inner Non-cacheable, Outer Non-cacheable, cache maintenance instructions that operate by VA to the PoPS affect the caches of other PEs in the shareability domain described by the shareability attributes of the VA supplied with the instruction.

For Device memory or Normal memory that is Inner Non-cacheable, Outer Non-cacheable, these instructions affect the caches of all PEs in the Outer Shareable shareability domain of the PE on which the instructions are operating.

In all cases, for any affected PE, these instructions affect all data and unified caches to the PoPS. Table D4-5 shows the scope of these Data and unified cache maintenance to the PoPS instructions.

Table D4-5 PEs affected by cache maintenance instructions to the PoPS

Shareability	PEs affected	Effective to
Non-shareable	The PE executing the instruction.	The PoPS of the entire system.
Inner Shareable	All PEs in the same Inner Shareable shareability domain as the PE executing the instruction.	The PoPS of the entire system.
Outer Shareable	All PEs in the same Outer Shareable shareability domain as the PE executing the instruction.	The PoPS of the entire system.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

It is IMPLEMENTATION DEFINED by the system whether the cache maintenance instructions have an effect on the caches of observers that are not PEs within the affected shareability domain to which the cache maintenance instructions apply.

D4.4.8.8 Effects of instructions that operate by VA to the PoU

For cache maintenance instructions that operate by VA to the PoU, Table D4-6 shows how, for a VA in a Normal or Device memory location, the shareability attribute of the VA determines the minimum set of PEs affected, and the point to which the instruction must be effective.

Table D4-6 PEs affected by cache maintenance instructions to the PoU

Shareability	PEs affected	Effective to
Non-shareable	The PE executing the instruction	The PoU of instruction cache fills, data cache fills and write-backs, and translation table walks, on the PE executing the instruction
Inner Shareable or Outer Shareable	All PEs in the same Inner Shareable shareability domain as the PE executing the instruction	The PoU of instruction cache fills, data cache fills and write-backs, and translation table walks, of all PEs in the same Inner Shareable shareability domain as the PE executing the instruction

Note

- The set of PEs guaranteed to be affected is never greater than the PEs in the Inner Shareable shareability domain containing the PE executing the instruction.
- It is IMPLEMENTATION DEFINED by the system whether the cache maintenance instructions have an effect on the caches of observers that are not PEs within the affected shareability domain to which the cache maintenance instructions apply.

D4.4.8.9 Effects of All and set/way maintenance instructions

The DC set/way instructions apply only to the caches of the PE that performs the instruction. IC IALLU instructions apply only to the caches of the PE that performs the instruction, unless HCR_EL2.FB=1, which causes the instructions to be broadcast within the Inner Shareable domain when executed from EL1.

The IC IALLUIS instruction can affect the caches of all PEs in the same Inner Shareable shareability domain as the PE that performs the instruction. This instruction has an effect to the Point of Unification of instruction cache fills, data cache fills, and write-backs, and translation table walks, of all PEs in the same Inner Shareable shareability domain.

Note

- The possible presence of system caches, as described in System level caches, means architecture does not guarantee that all levels of the cache can be maintained using set/way instructions.
- It is IMPLEMENTATION DEFINED by the system whether the cache maintenance instructions have an effect on the caches of observers that are not PEs within the affected shareability domain to which the cache maintenance instructions apply.

D4.4.8.10 Effects of virtualization and Security state on the cache maintenance instructions

Each Security state has its own physical address (PA) space, therefore cache entries are associated with PA space.

Table D4-7 shows the effects of virtualization and security on the cache maintenance instructions. In the table, the *Specified entries* are entries that the architecture requires the instruction to affect.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

The rules described in General behavior of the caches mean that an instruction might also affect other entries.

Table D4-7 Effects of virtualization and security on the maintenance instructions

Cache maintenance instructions	Security state	Specified entries
Data or unified cache maintenance instructions		
Invalidate, Clean, or Clean and Invalidate by VA: DC IVAC, DC CVAC, DC CVAP, DC CVAU, DC CIVAC, DC CVAP	Non-secure and Secure.	All lines that hold the PA that, in the current Security state, is mapped to by the combination of all of : • The specified VA. • For an instruction executed at EL0 the current VMID.[a]
Invalidate, Clean, or Clean and Invalidate by set/way: DC ISW, DC CSW, DCCISW	Non-secure	Line specified by set/way provided that the entry comes from the Non-secure PA space.
	Secure	Line specified by set/way provided that the entry comes from the Secure or Non-secure PA space.
Instruction cache maintenance instructions		
Invalidate by VA: IC IVAU	Non-secure and Secure.	All lines corresponding to the specified VA[b] in the current translation regime and: • For an instruction executed at EL0 the current VMID.[a]
Invalidate All: IC IALLU, IC IALLUIS	Non-secure and Secure.	For an instruction executed at: • EL1 all instruction cache lines containing Secure or Non-secure entries associated with the current VMID. • EL2 all instruction cache lines containing Non-secure entries.

a. Dependencies on the VMID apply even when HCR_EL2.VM is set to 0. The architecture does not define a reset value for VTTBR_EL2.VMID, and therefore, in any implementation that includes EL2, the boot software executed when reset is deasserted must initialize VTTBR_EL2.VMID.

b. The type of instruction cache used affects the interpretation of the specified entries in this table such that:

- For a PIPT instruction cache, the cache maintenance applies to all entries whose physical address corresponds to the specified address.
- For a VIPT instruction cache, the cache maintenance applies to entries whose virtual index and physical tag corresponds to the specified address.

For information on types of instruction cache see Instruction caches.

For locked entries and entries that might be locked, the behavior of cache maintenance instructions described in The interaction of cache lockdown with cache maintenance instructions applies.

With an implementation that generates aborts if entries are locked or might be locked in the cache, when the use of lockdown aborts is enabled, these aborts can occur on any cache maintenance instructions.

In an implementation that includes EL2:

- The architecture does not require cache cleaning when switching between virtual machines. Cache invalidation by set/way must not present an opportunity for one virtual machine to corrupt state associated with a second virtual machine. To ensure this requirement is met, invalidate by set/way instructions can, instead, perform a clean and invalidate by set/way.
- As described in The data cache maintenance instruction (DC), the AArch64 Data cache invalidate instructions, DC IVAC and DC ISW, when executed at EL1 and EL0,
  can be configured to perform a cache clean as well as a cache invalidation.
- TLB invalidate instructions and IC IALLU instructions executed at EL1 are broadcast across the Inner Shareable domain when all of the following are true:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— EL2 is implemented and enabled in the current Security state.
— The value of HCR_EL2.FB is 1.

For more information about the cache maintenance instructions, see About cache maintenance in AArch64 state, A64 Cache maintenance instructions, and The AArch64 Virtual Memory System Architecture.

D4.4.8.11 Boundary conditions for cache maintenance instructions

Cache maintenance instructions operate on the caches regardless of whether the System register Cacheability controls force all memory accesses to be Non-cacheable.

For VA-based cache maintenance instructions, the instruction operates on the caches regardless of the memory type and cacheability attributes marked for the memory address in the VMSA translation table entries. This means that the effects of the cache maintenance instructions can apply regardless of:

- Whether the address accessed:
  — Is Normal memory or Device memory.
  — Has the Cacheable attribute or the Non-cacheable attribute.
- Any applicable domain control of the address accessed.
- The access permissions for the address accessed, other than the effect of the stage two write permission on data or unified cache invalidation instructions.

D4.4.8.12 Ordering and completion of data and instruction cache instructions

All data cache instructions, other than DC ZVA that specify an address:

- Execute in program order relative to loads or stores that have all of the following properties:
  — Access an address in Normal memory with either Inner Write Through or Inner Write Back attributes within the same cache line of minimum size, as indicated by CTR_EL0.DMinLine.
  — Use an address with the same cacheability attributes as the address passed to the data cache instruction.
- Can execute in any order relative to loads or stores that have all of the following properties:
  — Access an address in Normal memory with either Inner Write Through or Inner Write Back attributes within the same cache line of minimum size, as indicated by CTR_EL0.DMinLine.
  — Use an address with different cacheability attributes as the address passed to the data cache instruction.
  — Do not have a `DMB` or `DSB` executed between the load or store instruction and the data cache instruction.
- Can execute in any order relative to loads or stores that access any address with the Device memory attribute, or with Normal memory with Inner Non-cacheable attribute unless a `DMB` or `DSB` is executed between the instructions.
- Execute in program order relative to other data cache instructions, other than DC ZVA that specify an address within the same cache line of minimum size, as indicated by CTR_EL0.DMinLine.
- Can execute in any order relative to loads or stores that access an address in a different cache line of minimum size, as indicated by CTR_EL0.DMinLine, unless a `DMB` or `DSB` is executed between the instructions.
- Can execute in any order relative to other data cache instructions, other than DC ZVA that specify an address in a different cache line of minimum size, as indicated by CTR_EL0.DMinLine, unless a `DMB` or `DSB` is executed between the instructions.
- Can execute in any order relative to instruction cache maintenance instructions unless a `DSB` is executed between the instructions.
- Can execute in any order relative to data cache maintenance instructions that do not specify an address unless a `DMB` or `DSB` is executed between the instructions.

Note

- Data cache ordering rules by address are consistent with physically indexed physically tagged caches. See Data and unified caches.
- Data cache zero instruction describes the ordering and completion rules for Data Cache Zero.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

All data cache maintenance instructions that do not specify an address:

- Can execute in any order relative to data cache maintenance instructions that do not specify an address unless a DMB or DSB is executed between the instructions.
- Can execute in any order relative to data cache maintenance instructions that specify an address, unless a DMB or DSB is executed between the instructions.
- Can execute in any order relative to loads or stores unless a DMB or DSB is executed between the instructions.
- Can execute in any order relative to instruction cache maintenance instructions unless a DSB is executed between the instructions.

All instruction cache maintenance instructions can execute in any order relative to other instruction cache instructions, data cache instructions, loads, and stores unless a DSB is executed between the instructions.

A cache maintenance instruction can complete at any time after it is executed, but is only guaranteed to be complete, and its effects visible to other observers, following a DSB instruction executed by the PE that executed the cache maintenance instruction. See also the requirements for cache maintenance instructions in Completion and endpoint ordering.

In all cases, where the text in this section refers to a DMB or a DSB, this means a DMB or DSB whose required access type is both loads and stores.

Note

This section specifies the memory ordering requirements of DMB and DSB based on the required access type as specified by the <option> parameter. DSB has additional requirements with respect to execution of instructions appearing in program order after the DSB as described in Data Synchronization Barrier. These additional requirements create the effect of additional memory ordering beyond what is specified in this section.

D4.4.8.12.1 Ordering and completion of Data Cache Clean to Point of Persistence

The update of the persistent memory as a result of Data Cache Clean to the Point of Persistence is guaranteed to have occurred either after:

- The execution of a DSB applying to both reads and writes after the execution of the Data Cache Clean to the Point of Persistence.
- The update to persistent memory caused by a different Data Cache Clean to the Point of Persistence that is ordered after a DMB applying to both reads and writes that appears after the original Data Cache Clean to the Point of Persistence.

Note

This second point is an aspect of the fact that the Data Cache Clean to the Point of Persistence instructions are ordered by DMB, and this controls the order of arrival in persistent memory.

The ordering effect for the Data Cache Clean to the Point of Persistence by DMB applying to both read and writes is not sufficient to ensure that in Example D4-1, observation of the value '1' in the memory location X3 implies that the Data Cache Clean to the Point of Persistence has caused an update of persistent memory:

Example D4-1 The ordering effect for the Data Cache Clean to the Point of Persistence

```
; initial condition has [X3]=0

DC CVADP, X1
DMB
MOV X2,#1
STR X2, [X3]
```

However, in Example D4-2, the ordering effects of the DMB instruction will ensure that the location pointed by P0: X1 will reach the Point of Persistence before, or at the same time as, the location pointed by P1:X8.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example D4-2 The ordering effect for the Data Cache Clean to the Point of Persistence

```
; initial conditions has P0: X3 and P1: X3 point to the same location, which is 0 at the start of
    ↪this example

P0
    DC CVAP, X1
    DMB
    MOV X2, #1
    STR X2, [X3]

P1
loop
    LDR X2, [X3]
    CBZ X2, loop
    DMB
    DC CVAP, X8
```

D4.4.8.12.2 Ordering and completion of Data Cache Clean to Point of Deep Persistence

The update of the deep persistent memory as a result of Data Cache Clean to the Point of Deep Persistence is guaranteed to have occurred either after:

- The execution of a DSB applying to both reads and writes after the execution of the Data Cache Clean to the Point of Deep Persistence.
- The update to deep persistent memory caused by a different Data Cache Clean to the Point of Deep Persistence that is ordered after a `DMB` applying to both reads and writes that appears after the original Data Cache Clean to the Point of Deep Persistence.

Note

This second point is an aspect of the fact that the Data Cache Clean to the Point of Deep Persistence instructions are ordered by DMB, and this controls the order of arrival in deep persistent memory.

The ordering effect for the Data Cache Clean to the Point of Deep Persistence by `DMB` applying to both read and writes is not sufficient to ensure that in Example D4-3, observation of the value '1' in the memory location X3 implies that the Data Cache Clean to the Point of Deep Persistence has caused an update of deep persistent memory:

Example D4-3 The ordering effect for the Data Cache Clean to the Point of Deep Persistence

```
; initial conditions has [X3]=0

DC CVADP, X1
DMB
MOV X2,#1
STR X2, [X3]
```

However, in Example D4-4, the ordering effects of the `DMB` instruction will ensure that the location pointed by P0: X1 will reach the Point of Deep Persistence before, or at the same time as, the location pointed by P1: X8.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example D4-4 The ordering effect for the Data Cache Clean to the Point of Deep Persistence

```
; initial conditions has P0: X3 and P1: X3 point to the same location, which is 0 at the start of
    ↪this example
P0
    DC CVADP, X1
    DMB
    MOV X2, #1
    STR X2, [X3]

P1
loop
    LDR X2, [X3]
    CBZ X2, loop
    DMB
    DC CVADP, X8
```

D4.4.8.13 Performing cache maintenance instructions

To ensure all cache lines in a block of address space are maintained through all levels of cache Arm strongly recommends that software:

- For data or unified cache maintenance, uses the CTR_EL0.DMinLine value to determine the loop increment size for a loop of data cache maintenance by VA instructions.
- For instruction cache maintenance, uses the CTR_EL0.IMinLine value to determine the loop increment size for a loop of instruction cache maintenance by VA instructions.

D4.4.8.13.1 Example code for cache maintenance instructions

The cache maintenance instructions by set/way can clean or invalidate, or both, the entirety of one or more levels of cache attached to a PE. However, unless all PEs attached to the caches regard all memory locations as Non-cacheable, it is not possible to prevent locations being allocated into the cache during such a sequence of the cache maintenance instructions.

Note

Since the set/way instructions are performed only locally, there is no guarantee of the atomicity of cache maintenance between different PEs, even if those different PEs are each executing the same cache maintenance instructions at the same time. Since any cacheable line can be allocated into the cache at any time, it is possible for a cache line to migrate from an entry in the cache of one PE to the cache of a different PE in a way that means the line is not affected by set/way based cache maintenance. Therefore, Arm strongly discourages the use of set/way instructions to manage coherency in coherent systems. The expected use of the cache maintenance instructions that operate by set/way is limited to the cache maintenance associated with the powerdown and powerup of caches, if this is required by the implementation.
The limitations of cache maintenance by set/way mean maintenance by set/way does not happen on multiple PEs, and cannot be made to happen atomically for each address on each PE. Therefore in multiprocessor or multithreaded systems, the use of cache maintenance by set/way to clean, or clean and invalidate, the entire cache for coherency management with very large buffers or with buffers with unknown address can fail to provide the expected coherency results because of speculation by other PEs, or possibly by other threads. The only way that these instructions can be used in this way is to first ensure that all PEs that might cause speculative accesses to caches that need to be maintained are not capable of generating speculative accesses. This can be achieved by ensuring that those PEs have no memory locations with a Normal Cacheable attribute. Such an approach can have very large system performance effects, and Arm advises implementers to use hardware coherency mechanisms in systems where this will be an issue.
System level caches refers to other limitations of cache maintenance by set/way.

The following example code for cleaning a data or unified cache to the Point of Coherency illustrates a generic mechanism for cleaning the entire data or unified cache to the Point of Coherency. It assumes that the current Cache Size Identification Register is in 32-bit format. For more information, see Possible formats of the Cache Size Identification Register, CCSIDR_EL1.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        MRS     X0, CLIDR_EL1
        AND     W3, W0, #0x07000000     // Get 2 x Level of Coherence
        LSR     W3, W3, #23
        CBZ     W3, Finished
        MOV     W10, #0                 // W10 = 2 x cache level
        MOV     W8, #1                  // W8 = constant 0b1
Loop1:  ADD     W2, W10, W10, LSR #1    // Calculate 3 x cache level
        LSR     W1, W0, W2              // extract 3-bit cache type for this level
        AND     W1, W1, #0x7
        CMP     W1, #2
        B.LT    Skip                    // No data or unified cache at this level
        MSR     CSSELR_EL1, X10         // Select this cache level
        ISB                             // Synchronize change of CSSELR
        MRS     X1, CCSIDR_EL1          // Read CCSIDR
        AND     W2, W1, #7              // W2 = log2(linelen)-4
        ADD     W2, W2, #4              // W2 = log2(linelen)
        UBFX    W4, W1, #3, #10         // W4 = max way number, right aligned
        CLZ     W5, W4                  // W5 = 32-log2(ways), bit position of way in DC operand
        LSL     W9, W4, W5              // W9 = max way number, aligned to position in DC operand
        LSL     W16, W8, W5             // W16 = amount to decrement way number per iteration
Loop2:  UBFX    W7, W1, #13, #15        // W7 = max set number, right aligned
        LSL     W7, W7, W2              // W7 = max set number, aligned to position in DC operand
        LSL     W17, W8, W2             // W17 = amount to decrement set number per iteration
Loop3:  ORR     W11, W10, W9            // W11 = combine way number and cache number ...
        ORR     W11, W11, W7            //  ... and set number for DC operand
        DC      CSW, X11                // Do data cache clean by set and way
        SUBS    W7, W7, W17             // Decrement set number
        B.GE    Loop3
        SUBS    X9, X9, X16             // Decrement way number
        B.GE    Loop2
Skip:   ADD     W10, W10, #2            // Increment 2 x cache level
        CMP     W3, W10
        DSB                             // Ensure completion of previous cache maintenance instruction
        B.GT    Loop1
Finished:
```

Similar approaches can be used for all cache maintenance instructions.

D4.4.9 Data cache zero instruction

The Data Cache Zero by Address instruction, DC ZVA, writes `0x00` to each byte of a block of *N* bytes, aligned in memory to *N* bytes in size, where:

- The block in memory is identified by the address supplied as an argument to the DC ZVA instruction. There are no alignment restrictions on this address.

  Note

  This means that each byte of the block of memory that includes the supplied address is set to zero.

- The DCZID_EL0 register indicates the block size, *N* bytes, that is written with byte values of zero.

Software can restrict access to this instruction. See Configurable instruction controls and the description of the DC ZVA instruction.

The DC ZVA instruction behaves as a set of stores to the location being accessed, and:

- Generates a Permission fault if the translation regime being used when the instruction is executed does not permit writes to the locations.
- Requires the same considerations for ordering and the management of coherency as any other store instruction.

In addition:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When the instruction is executed, it can generate memory faults or watchpoints that are prioritized in the same way as other memory related faults or watchpoints. Where a synchronous Data Abort fault or a watchpoint is generated, the CM bit in the syndrome field is not set to 1, which would be the case for all other cache maintenance instructions. See ISS encoding for an exception from a Data Abort for more information about the encoding of the associated ESR_ELx.ISS field.
- If the memory region being zeroed is any type of Device memory, then `DC ZVA` generates an Alignment fault which is prioritized in the same way as other alignment faults that are determined by the memory type.

Note

The architecture makes no statements about whether or not a DC ZVA instruction causes allocation to any particular level of the cache, for addresses that have a cacheable attribute for those levels of cache.
Despite its mnemonic, the DC ZVA instruction is not a data cache maintenance instruction.

D4.4.10 Cache lockdown

The concept of an entry locked in a cache is allowed, but not architecturally defined. How lockdown is achieved is IMPLEMENTATION DEFINED and might not be supported by:

- An implementation.
- Some memory attributes.

An unlocked entry in a cache might not remain in that cache. The architecture does not guarantee that an unlocked cache entry remains in the cache or remains incoherent with the rest of memory. Software must not assume that an unlocked item that remains in the cache remains dirty.

A locked entry in a cache is guaranteed to remain in that cache. The architecture does not guarantee that a locked cache entry remains incoherent with the rest of memory, that is, it might not remain dirty.

D4.4.10.1 The interaction of cache lockdown with cache maintenance instructions

The interaction of cache lockdown and cache maintenance instructions is IMPLEMENTATION DEFINED. However, an architecturally-defined cache maintenance instruction on a locked cache line must comply with the following general rules:

- The effect of the following instructions on locked cache entries is IMPLEMENTATION DEFINED:
  - Cache clean by set/way, DC CSW.
  - Cache invalidate by set/way, DC ISW.
  - Cache clean and invalidate by set/way, DC CISW.
  - Instruction cache invalidate all, IC IALLU and IC IALLUIS.

  However, one of the following approaches must be adopted in all these cases:

  1. If the instruction specified an invalidation, a locked entry is not invalidated from the cache.
  2. If the instruction specified a clean it is IMPLEMENTATION DEFINED whether locked entries are cleaned.
  3. If an entry is locked down, or could be locked down, an IMPLEMENTATION DEFINED Data Abort exception is generated, using the DFSC value defined for this purpose, see ISS encoding for an exception from a Data Abort.

  This permits a usage model for cache invalidate routines to operate on a large range of addresses by performing the required operation on the entire cache, without having to consider whether any cache entries are locked.
- The effect of the following instructions on locked cache entries is IMPLEMENTATION DEFINED:
  - Cache clean by virtual address, DC CVAC, DC CVAP, and DC CVAU.
  - Cache invalidate by virtual address, DC IVAC.
  - Cache clean and invalidate by virtual address, DC CIVAC.

  However, one of the following approaches must be adopted in all these cases:

  1. If the instruction specified an invalidation, a locked entry is invalidated from the cache. For the clean and invalidate instructions, the entry must be cleaned before it is invalidated.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2. If the instruction specified an invalidation, a locked entry is not invalidated from the cache. If the instruction specified a clean it is IMPLEMENTATION DEFINED whether locked entries are cleaned.
3. If an entry is locked down, or could be locked down, an IMPLEMENTATION DEFINED Data Abort exception is generated, using the DFSC value defined for this purpose. See ESR_ELx.

In an implementation that includes EL2 enabled in the current Security state, if HCR_EL2.TIDCP is set to 1, any exception relating to lockdown of an entry is routed to EL2.

Note

An implementation that uses an abort mechanism for entries that can be locked down but are not actually locked down must:

- Document the IMPLEMENTATION DEFINED instruction sequences that perform the required operations on entries that are not locked down.
- Implement one of the other permitted alternatives for the locked entries.

Arm recommends that, when possible, such IMPLEMENTATION DEFINED instruction sequences use architecturally-defined instructions. This minimizes the number of customized instructions required.
In addition, an implementation that uses an abort to handle cache maintenance instructions for entries that might be locked must provide a mechanism that ensures that no entries are locked in the cache.
The reset setting of the cache must be that no cache entries are locked.

D4.4.10.1.1 Additional cache functions for the implementation of lockdown

An implementation can add additional cache maintenance functions for the handling of lockdown in the IMPLEMENTATION DEFINED spaces reserved for Cache Lockdown, see Reserved encodings for IMPLEMENTATION DEFINED registers.

D4.4.11 System level caches

The Arm Architecture defines a *system cache* as a cache that is not described in the PE Cache Identification registers, CCSIDR_EL1 and CLIDR_EL1, and for which the set/way cache maintenance instructions do not apply.

Conceptually, three classes of system cache can be envisaged:

1. System caches which lie before the Point of Coherency and cannot be managed by any cache maintenance instructions. Such systems fundamentally undermine the concept of cache maintenance instructions operating to the Point of Coherency, as they imply the use of non-architecture mechanisms to manage coherency. The use of such systems in the Arm architecture is explicitly prohibited.
2. System caches which lie before the Point of Coherency and can be managed by cache maintenance by address instructions that apply to the Point of Coherency, but cannot be managed by cache maintenance by set/way instructions. Where maintenance of the entirety of such a cache must be performed, as in the case for power management, it must be performed using non-architectural mechanisms.
3. System caches which lie beyond the Point of Coherency and so are invisible to the software for the purpose of managing coherency.

Note

If maintenance of the entirety of such a cache is supported, it is an IMPLEMENTATION DEFINED property of the system.

D4.4.12 Branch prediction

In the AArch64 architecture, branch predictors are not architecturally visible.

For details on the restrictions on the effects of speculation for branch predictors, see Restrictions on the effects of speculation.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D4.5 External aborts

The Arm architecture defines External aborts as errors that occur in the memory system, other than those that are detected by the MMU, MPU, or debug logic. An External abort might signal a data corruption to the PE. For example, a memory location might have been corrupted, and this corruption is detected by hardware using a parity or error correction code (ECC). The error might have been propagated. The RAS Extension provides mechanisms for software to determine the extent of the corruption and contain propagation of the error. For more information, see RAS PE Architecture.

An External abort is one of the following:

- Synchronous.
- Precise asynchronous.
- Imprecise asynchronous.

For more information, see Exception entry terminology.

The RAS Extension provides a more granular taxonomy of aborts. When the RAS Extension is not implemented, the Arm architecture does not provide any method to distinguish between precise asynchronous and imprecise asynchronous External aborts.

It is IMPLEMENTATION DEFINED which External aborts, if any, are supported.

External aborts on data accesses and translation table walks on data accesses can be either synchronous or asynchronous.

When FEAT_DoubleFault is not implemented, External aborts on instruction fetches and translation table walks on instruction fetches can be either synchronous or asynchronous.

When FEAT_DoubleFault is implemented, all External abort exceptions on instruction fetches and translation table walks on instruction fetches must be synchronous.

A synchronous External abort on an instruction fetch, including a translation table walk on an instruction fetch, is taken precisely using the Instruction Abort exception.

A synchronous External abort on a data read or write, including a translation table walk on a data read or write, is taken precisely using the Data Abort exception.

See Synchronous exception types.

An asynchronous External abort is taken using the SError exception. See Asynchronous exception types.

The effect of a failed memory access is described in Definition of a precise exception and imprecise exception.

Normally, External aborts are rare. An imprecise asynchronous External abort is likely to be fatal to the process that is running, Arm recommends that implementations make External aborts precise wherever possible.

The following subsections give more information about possible External aborts:

- Provision for the classification of External aborts.
- Parity or ECC error reporting, RAS Extension not implemented.

D4.5.1 Provision for the classification of External aborts

In AArch64 state, an implementation can use ESR_ELx.EA, ISS[9], to provide more information about synchronous External aborts. For all synchronous aborts other than synchronous External aborts, ESR_ELx.EA, ISS[9], returns a value of 0.

If the FEAT_RAS is implemented:

- The ESR_ELx.SET field provides information about the state of the PE following a synchronous External abort.
- The ESR_ELx.AET field might contain more information following an asynchronous abort taken as an SError exception.
- The implementation might define error record registers.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For more information, see:

- ISS encoding for an exception from an Instruction Abort.
- ISS encoding for an exception from a Data Abort.
- ISS encoding for an SError exception.
- Taking error exceptions.

D4.5.2 Parity or ECC error reporting, RAS Extension not implemented

The Arm architecture supports the reporting of both synchronous and asynchronous parity or ECC errors from the cache system. It is IMPLEMENTATION DEFINED what parity or ECC errors in the cache systems, if any, result in synchronous or asynchronous parity or ECC errors.

A fault code is defined for reporting parity or ECC errors. However, when parity or ECC error reporting is implemented, it is IMPLEMENTATION DEFINED whether a parity or ECC error is reported using the assigned fault code or using another appropriate encoding.

For all purposes other than the Fault status encoding, parity or ECC errors are treated as External aborts.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D4.6 Memory barrier instructions

Memory barriers describes the memory barrier instructions. This section describes the system level controls of those instructions.

D4.6.1 EL2 control of the shareability of data barrier instructions executed at EL0 or EL1

In an implementation that includes EL2 enabled in the current Security state and supports shareability limitations on the data barrier instructions, the HCR_EL2.BSU field can modify the required shareability of an instruction that is executed at EL0 or EL1. Table D4-8 shows the encoding of this field.

Table D4-8 EL2 control of shareability of barrier instructions executed at EL0 or EL1

HCR_EL2.BSU	Minimum shareability of barrier instructions
00	No effect, shareability is as specified by the instruction
01	Inner Shareable
10	Outer Shareable
11	Full system

For an instruction executed at EL0 or EL1, Table D4-9 shows how the HCR_EL2.BSU is combined with the shareability specified by the argument of the `DMB` or `DSB` instruction to give the scope of the instruction.

Table D4-9 Effect of HCR_EL2.BSU on barrier instructions executed at EL1 or EL0

Shareability specified by the `DMB` or `DSB` argument	HCR_EL2.BSU	Resultant shareability
Full system	Any	Full system
Outer Shareable	`00`, `01`, or `10`	Outer Shareable
	`11`, Full system	Full system
Inner Shareable	`00` or `01`	Inner Shareable
	`10`, Outer Shareable	Outer Shareable
	`11`, Full system	Full system
Non-shareable	`00`, No effect	Non-shareable
	`01`, Inner Shareable	Inner Shareable
	`10`, Outer Shareable	Outer Shareable
	`11`, Full system	Full system

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D4.7 Pseudocode description of general memory System instructions

This section lists the pseudocode describing general memory operations:

- Memory data type definitions.
- Basic memory access.
- Aligned memory access.
- Unaligned memory access.
- Exclusives monitors operations.
- Access permission checking.
- Abort exceptions.
- Memory barriers.

D4.7.1 Memory data type definitions

This section lists the memory data types.

The memory data types are:

- Address descriptor, defined by the `AddressDescriptor` type.
- Full address, defined by the `FullAddress` type.
- Memory attributes, defined by the `MemoryAttributes` type.
- Memory type, defined by the `MemType` enumeration.
- Device memory type, defined by the `DeviceType` enumeration.
- Normal memory attributes, defined by the `MemAttrHints` type.
- Cacheability attributes, defined by the `MemAttr_NC`, `MemAttr_WT`, and `MemAttr_WB` constants.
- Allocation hints, defined by the `MemHint_No`, `MemHint_WA`, `MemHint_RA`, and `MemHint_RWA` constants.
- Access permissions, defined by the `Permissions` type.

These types are defined in A-profile Architecture Pseudocode.

D4.7.2 Basic memory access

The `PhysMemRead()` and `PhysMemWrite()` functions perform single-copy atomic, aligned, little-endian memory accesses of `size` bytes to or from the underlying physical memory array of bytes.

The attributes in `memaddrdesc.memattrs` are used by the memory system to determine caching and ordering behaviors as described in Memory types and attributes, Ordering relations, and Atomicity in the Arm architecture.

D4.7.3 Aligned memory access

The two `MemSingle[]` accessors, non-assignment (memory read) `AArch64.MemSingle[]` and assignment (memory write) `AArch64.MemSingle[]`, make atomic, little-endian accesses of `size` bytes.

D4.7.4 Unaligned memory access

The two `Mem[]` accessors, Non-assignment (memory read) `Mem[]` and Assignment (memory write) `Mem[]`, make accesses of the required type. If an access is not architecturally defined to be atomic, `Mem[]` synthesizes accesses from multiple calls to `AArch64.MemSingle[]`. It also reverses the byte order if the access is big-endian.

The `IsAligned()` function checks the alignment of memory accesses.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D4.7.5 Exclusives monitors operations

The `AArch64.SetExclusiveMonitors()` function sets the Exclusives monitors for a block of bytes, the size of which is determined by `size`, at the virtual address defined by `address`.

The `AArch64.ExclusiveMonitorsPass()` function checks whether the Exclusives monitors are set to include the location of a number of bytes specified by `size`, at the virtual address defined by `address`. The atomic write that follows after the Exclusives monitors have been set must be to the same physical address. It is permitted, but not required, for this function to return FALSE if the virtual address is not the same as that used in the previous call to `AArch64.SetExclusiveMonitors()`.

The `ExclusiveMonitorsStatus()` function returns 0 if the previous atomic write was to the same physical memory locations selected by `AArch64.ExclusiveMonitorsPass()` and therefore succeeded. Otherwise the function returns 1, indicating that the address translation delivered a different physical address.

The `MarkExclusiveGlobal()` procedure takes as arguments a `FullAddress paddress`, the PE identifier `processorid` and the size of the transfer. The procedure records that the PE `processorid` has requested exclusive access covering at least `size` bytes from address `paddress`. The size of the location marked as exclusive is IMPLEMENTATION DEFINED, up to a limit of 2KB and no smaller than two words, and aligned in the address space to the size of the location. It is CONSTRAINED UNPREDICTABLE whether this causes any previous request for exclusive access to any other address by the same PE to be cleared.

The `MarkExclusiveLocal()` procedure takes as arguments a `FullAddress paddress`, the PE identifier `processorid` and the size of the transfer. The procedure records in a local record that PE `processorid` has requested exclusive access to an address covering at least `size` bytes from address paddress. The size of the location marked as exclusive is IMPLEMENTATION DEFINED, and can at its largest cover the whole of memory but is no smaller than two words, and is aligned in the address space to the size of the location. It is IMPLEMENTATION DEFINED whether this procedure also performs a `MarkExclusiveGlobal()` using the same parameters.

The `IsExclusiveGlobal()` function takes as arguments a `FullAddress paddress`, the PE identifier `processorid` and the size of the transfer. The function returns TRUE if the PE `processorid` has marked in a global record an address range as exclusive access requested that covers at least `size` bytes from address `paddress`. It is IMPLEMENTATION DEFINED whether it returns TRUE or FALSE if a global record has marked a different address as exclusive access requested. If no address is marked in a global record as exclusive access, `IsExclusiveGlobal()` returns FALSE.

The `IsExclusiveLocal()` function takes as arguments a `FullAddress paddress`, the PE identifier `processorid` and the size of the transfer. The function returns TRUE if the PE `processorid` has marked an address range as exclusive access requested that covers at least the `size` bytes from address `paddress`. It is IMPLEMENTATION DEFINED whether this function returns TRUE or FALSE if the address marked as exclusive access requested does not cover all of `size` bytes from address `paddress`. If no address is marked as exclusive access requested, then this function returns FALSE. It is IMPLEMENTATION DEFINED whether this result is ANDed with the result of `IsExclusiveGlobal()` with the same parameters.

The `ClearExclusiveByAddress()` procedure takes as arguments a `FullAddress paddress`, the PE identifier `processorid` and the size of the transfer. The procedure clears the global records of all PEs, other than `processorid`, for which an address region including any of `size` bytes starting from `paddress` has had a request for an exclusive access. It is IMPLEMENTATION DEFINED whether the equivalent global record of the PE `processorid` is also cleared if any of `size` bytes starting from `paddress` has had a request for an exclusive access, or if any other address has had a request for an exclusive access.

The `ClearExclusiveLocal()` procedure takes as arguments the PE identifier `processorid`. The procedure clears the local record of PE `processorid` for which an address has had a request for an exclusive access. It is IMPLEMENTATION DEFINED whether this operation also clears the global record of PE `processorid` that an address has had a request for an exclusive access.

D4.7.6 Access permission checking

The AArch64.S1CheckPermissions() and AArch64.S2CheckPermissions() functions are used by the architecture to perform access permission checking based on attributes derived from the Translation Tables descriptors.

The interpretation of access permission is shown in Memory access control.

D4.7.7 Abort exceptions

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The `AArch64.Abort()` function generates either a Data Abort or an Instruction Abort exception by calling `AArch64.DataAbort()` or `AArch64.InstructionAbort()`. It also can generate a debug exception for debug related faults, see AArch64 Self-hosted Debug.

The `AArch64.DataAbort()` function generates a Data Abort exception, routes the exception to EL2, and records the information required for the Exception Syndrome registers, ESR_ELx. See ISS encoding for an exception from a Data Abort. A second stage abort might also record the intermediate physical address, IPA, but this depends on the type of the abort.

For a synchronous abort, `AArch64.DataAbort()` also sets the FAR to the VA of the abort.

The `AArch64.InstructionAbort()` function generates an Instruction Abort exception, routes the exception to EL2, and records the information required for the Exception Syndrome registers, ESR_ELx, see ISS encoding for an exception from an Instruction Abort. A second stage abort might also record the intermediate physical address, IPA, but this depends on the type of the abort.

For a synchronous abort, `AArch64.InstructionAbort()` also sets the FAR to the VA of the abort.

The `FaultRecord` type describes a fault. Functions that check for faults return a record of this type appropriate to the type of fault.

The function `NoFault()` returns a null record that indicates no fault. The `IsFault()` function tests whether a `FaultRecord` contains a fault.

D4.7.8 Memory barriers

The definition for the memory barrier functions is given by the enumerations `MBReqDomain` and `MBReqTypes`.

These enumerations define the required shareability domains and required access types used as arguments for `DMB` and `DSB` instructions.

The procedures `DataMemoryBarrier`, `DataSynchronizationBarrier`, and `InstructionSynchronizationBarrier` perform the memory barriers.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter D5
The AArch64 Virtual Memory System Architecture

This chapter provides a system level view of the AArch64 *Virtual Memory System Architecture* (VMSA), the memory system architecture of an A-profile implementation that is executing in AArch64 state. It contains the following sections:

- Address translation.
- Translation process.
- Translation table descriptor formats.
- Memory access control.
- Memory region attributes.
- Other descriptor fields.
- Address tagging.
- Pointer authentication.
- Memory aborts.
- Translation Lookaside Buffers.
- TLB maintenance.
- Caches.
- Pseudocode description of VMSAv8-64 address translation.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.1 Address translation

I_{BGZDZ} Armv8-R AArch64 supports VMSAv8-64 as an optional memory system architecture at stage 1 of the EL1&0 translation regime, and supports general-purpose operating systems, such as Linux and Android at EL1. With VMSAv8-64 supported at EL1 the Armv8-R AArch64 architecture profile can have the following memory system configurations:

- PMSAv8-64 at EL1 and EL2.
- PMSAv8-64 or VMSAv8-64 at EL1 and PMSAv8-64 at EL2.

I_{QTYRQ} The MSA and MSA_frac fields in ID_AA64MMFR0_EL1 identify the memory system configurations supported at EL1.

R_{BYTPV} All translation regimes in the R-profile VMSAv8-64 are Secure.

R_{YLPHH} In the Armv8-R AArch64 architecture the only permitted value of ID_AA64MMFR0_EL1.MSA is `0b1111`. When ID_AA64MMFR0_EL1.MSA_frac is `0b0010`, the stage 1 of the EL1&0 translation regime can enable the PMSAv8-64 or VMSAv8-64 architecture.

R_{VFJNJ} It is IMPLEMENTATION DEFINED whether a physical location is visible to a VMSAv8-64 context and to page table accesses, and it is permissible for an implementation to raise an External abort in this case.

I_{CCTQS} Address translation converts the addresses used by instructions to the addresses used by the physical memory system.

R_{CRPHV} When a data address or instruction address is used in an instruction, it is a *virtual address* (VA). This includes any address stored in one of the following registers:

- *Program counter* (PC).
- *Stack pointers* (SP).
- *Link register* (LR).
- *Exception link register* (ELR).

R_{CRDGS} When an access is made to the physical memory system, a *physical address* (PA) is used.

R_{KCNRX} An address translation maps a VA to a PA.

R_{ZKJWW} An address translation stage maps an *input address* (IA) to an *output address* (OA).

R_{BCKLH} If one address translation stage is used, then a VA is mapped to a PA using all of the following steps:

1. The VA is input as the IA to the translation stage.
2. The PA is output as the OA from the translation stage.

R_{HLQJT} The Armv8-R AArch64 VMSAv8-64 architecture does not support a VMSAv8-64 stage 2 translation regime. Memory accesses that would be subject to a stage 2 translation are managed by the PMSAv8-64 and is implemented by the EL2 MPU. References to the stage 2 translation refer to the permissions checks performed by the EL2 MPU.
If two address translation stages are used, then a VA is mapped to a PA using all of the following steps:

1. The VA is input as the IA to the stage 1 translation.
2. The *intermediate physical address* (IPA) is output as the OA from the stage 1 translation.
3. The IPA is input as the IA to the stage 2 translation, a permissions check by the EL2 MPU.
4. The memory access is either permitted by the EL2 MPU or a fault is generated.

For more information, see The AArch64 Protected Memory System Architecture.

I_{MBNVX} If an address translation stage is disabled, then the value of the OA is the same as the IA.

I_{YXNZL} When an IA is translated to an OA, using VMSAv8-64, an address translation stage uses a set of memory mapped translation tables that hold all of the following information:

- The OA corresponding to the IA.
- For accesses made from Secure state, whether the OA access is to the Secure or Non-secure address map.
- The OA memory access permissions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The OA memory region attributes.

I_{FJFQR} When an IA is translated by an address translation stage, all of the following apply:

- A translation table lookup reads an entry from a translation table.
- A translation table entry resolves a subset of the IA.
- Multiple translation table entries can be required to completely resolve an IA.
- An address translation can require multiple lookups across different lookup levels and multiple translation tables.

R_{JTYJP} When memory is accessed, the *Memory Management Unit* (MMU) controls address translation, memory access permissions, memory attribute determination, and memory attribute checking.

R_{RJPRG} When the MMU cannot translate the IA, an MMU fault is generated.

R_{ZCVCZ} When a stage 1 translation cannot translate an IA, a stage 1 MMU fault is generated.

I_{ZWCKD} For more information, see:

- Implemented physical address size.
- Output address size configuration.
- Supported virtual address ranges.
- Input address size configuration.
- Intermediate physical address size configuration.
- Translation process.
- The effects of disabling an address translation stage.
- Memory aborts
- The AArch64 Protected Memory System Architecture.

D5.1.1 Translation granules

I_{GZDPY} The translation granule size determines the number of bits resolved by each lookup level when mapping from an IA to OA, and the maximum size of a single translation table.

R_{ZVQXW} The VMSA defines all of the following translation granule sizes:

- 4KB.
- 16KB.
- 64KB.

R_{JZCBC} For each translation granule size, and translation stage, support is IMPLEMENTATION DEFINED.

R_{FZCLK} The stage 1 translation can be configured to use one of the supported translation granule sizes.

R_{QKCFY} A page is the smallest memory region in which an IA to OA mapping can be specified.

I_{MGPLT} The translation granule determines all of the following:

- The translation process used to resolve an IA to an OA.
- The page size of the address translation stage.
- The number of address bits required to address a page.
- The maximum translation table size of the address translation stage.
- The number of address bits that can be resolved in a single translation table lookup.

I_{KZLYC} For all of the following reasons, a larger translation granule can reduce the number of translation lookup levels:

- The larger granule uses a larger translation table with more entries.
- A single lookup can resolve more IA bits.
- The larger page size means more of the least significant address bits are used to address within a page and do not require translation because those bits are flat-mapped from IA to OA.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{QRLDB} Arm recommends separating memory-mapped peripherals by an integer multiple of the largest granule size supported by the PEs in the system, to allow independent management of each peripheral.

D5.1.2 Translation regimes

I_{CJMWY} A translation regime determines how a VA is mapped to a PA. The translation regime is affected by the current Security state, the current Exception level, the enabled Exception levels, HCR_EL2 settings, and implemented features.

I_{QQMYL} If a software agent, such as an operating system, uses or configures stage 1 translations, then that software agent might be unaware of all of the following:

- The EL2 MPU.
- The distinction between IPA and PA.

D5.1.2.1 Secure EL1&0 translation regime

R_{DKYXN} If one of the following applies, then stage 1 of the Secure EL1&0 translation regime is used to translate addresses:

- Memory is accessed from EL1.
- Memory is accessed from EL0 and the Effective value of HCR_EL2.TGE is 0.

R_{NRSBW} The Secure EL1&0 translation regime, supports all of the following:

- Translation stage 1 that does all the following:
  - Maps VAs to IPAs.
  - Supports two VA ranges and the use of ASIDs.

For more information, see Use of ASIDs and VMIDs to reduce TLB maintenance requirements.

R_{XMCNY} If the PE supports both PMSAv8-64 and VMSAv8-64 at EL1, then VTCR_EL2.MSA determines the memory system architecture enabled at stage 1 of the Secure EL1&0 translation regime. Depending on the enabled memory system architecture, stage 1 of the Secure EL1&0 translation regime is controlled by either of the following:

- An EL1 MPU for PMSAv8-64.
- An MMU for VMSAv8-64.

D5.1.2.2 Secure EL2 translation regime

R_{MBXNP} The stage 2 of the Secure EL1&0 translation regime and the stage 1 of the Secure EL2 translation regime are controlled by the EL2 MPU. Armv8-R AArch64 uses the same translation format and Fault encodings as the VMSAv8-64.

D5.1.3 The system registers relevant to MMU operation

I_{RJZKJ} If a System register name has a numeric suffix, then the suffix indicates the lowest Exception level that can access the register.

I_{HZYYY} Translation stages are enabled and controlled by the registers specified in the following table.

Table D5-1 Registers that enable and control translation stages

Translation Stage	Controlled from	Controlling registers
EL1&0 stage 1	EL1	SCTLR_EL1, TCR_EL1, MAIR_EL1, AMAIR_EL1, TTBR0_EL1, TTBR1_EL1, HCR_EL2

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{LBPSZ} The following table lists the System register common abbreviations used in VMSAv8-64. The common abbreviation is used to describe features that apply to multiple translation regimes or translation stages.

Table D5-2 Common System register abbreviations used in VMSAv8-64

Common Abbreviation	Translation stage	Exception level EL1	EL2
AMAIR_ELx	-	AMAIR_EL1	AMAIR_EL2
MAIR_ELx	-	MAIR_EL1	MAIR_EL2
SCTLR_ELx	-	SCTLR_EL1	SCTLR_EL2
TCR_ELx	Stage 1	TCR_EL1	TCR_EL2
	Stage 2		VTCR_EL2, VSTCR_EL2
TTBR_ELx	Stage 1	TTBR0_EL1, TTBR1_EL1	
TTBRn_ELx	Stage 1	TTBR0_EL1, TTBR1_EL1	
TTBR0_ELx	Stage 1	TTBR0_EL1	
TTBR1_ELx	Stage 1	TTBR1_EL1	

I_{RGGJW} When software that enables or disables an address translation stage uses a PA that differs from the VA, speculative instruction fetching can cause complications. If software controls address translations that apply to the software itself, then when that software enables or disables an address translation stage, Arm recommends that the software uses a PA that is identical to the VA.

I_{DLLGV} For more information, see:

- Register name disambiguation by Exception level.

D5.1.4 Out-of-context translation regimes

I_{PTDBT} Software is required to consider the effects on registers and translation tables when a context switch is made from one translation regime to another.

D5.1.4.1 Speculative memory accesses from out-of-context translation regimes

R_{NRJPP} If an implementation is executing at EL2, the PE is not permitted to use the registers associated with the EL1&0 translation regime to speculatively access memory or translation tables.

R_{LFHQG} When an exception to a higher Exception level occurs, memory accesses caused by the translation table walk for the lower exception level can continue speculatively after the higher Exception level is entered.

I_{QVPYJ} When an exception to a higher Exception level occurs, executing a DSB instruction after entering that higher Exception level ensures that the memory accesses caused by the translation table walk are completed.

D5.1.4.2 Register changes when changing virtual machine

I_{XHBRX} If EL2 is enabled, when a switch from one virtual machine to another occurs, software executing at EL0 or EL1 expects register fields controlling or affecting the EL1&0 regime are modified consistently before the next virtual machine starts execution.

I_{SFZVV} If the registers fields controlling or affecting the EL1&0 translation regime are modified when changing a virtual machine, then all of the following apply:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Software executing at EL2 does the modification.
- The registers are modified out-of-context from software executing at EL1 or EL0.
- When modifying the registers, no synchronization precautions are required until the exception return to EL1 or EL0.

D5.1.5 Implemented physical address size

I_{FRVXB} The implemented PA size is indicated by the value of the ID_AA64MMFR0_EL1.PARange field, as shown in the following table. All other values of the ID_AA64MMFR0_EL1.PARange field are reserved.

Table D5-3 Implemented physical address size

ID_AA64MMFR0_EL1.PARange	PA memory size	PA address size
0b0000	4GB	32 bits, PA[31:0]
0b0001	64GB	36 bits, PA[35:0]
0b0010	1TB	40 bits, PA[39:0]
0b0011	4TB	42 bits, PA[41:0]
0b0100	16TB	44 bits, PA[43:0]
0b0101	256TB	48 bits, PA[47:0]
0b0110	4PB	52 bits, PA[51:0]

R_{HGNFW} If the implemented PA size is greater than 48 bits, then FEAT_LPA is required.

D5.1.6 Output address size configuration

R_{VRZGZ} For a translation stage that uses the VMSAv8-64 translation system, the maximun OA size that can be described by translation table entries is one of the following:

- If FEAT_LPA is not implemented, then 48 bits.
- If FEAT_LPA is implemented, then 52 bits.

I_{XSLRQ} When VTCR_EL2.MSA is 1, the OA size from a translation stage is configured by the associated TCR_EL1.IPS field, as shown in the following table.

Table D5-4 Output address size configuration

TCR_EL1.IPS	OA memory size	OA address size
0b000	4GB	32 bits, OA[31:0]
0b001	64GB	36 bits, OA[35:0]
0b010	1TB	40 bits, OA[39:0]
0b011	4TB	42 bits, OA[41:0]
0b100	16TB	44 bits, OA[43:0]
0b101	256TB	48 bits, OA[47:0]
0b110	4PB	52 bits, OA[51:0]
0b111	64PB	56 bits, OA[55:0]

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{MNZFV} The TCR_EL1.IPS field is three bits and has the same encoding as the three least significant bits of ID_AA64MMFR0_EL1.PARange.

R_{CLPQT} The OA size from a translation stage cannot be larger than the implemented PA size. If the OA size represented by TCR_EL1.IPS is larger than the implemented PA size, then the OA size is treated as being the same as the implemented PA size.

I_{PKBTM} Arm strongly recommends that software avoids configuring TCR_EL1.IPS to a value greater than the implemented PA size.

I_{HGDWP} If a translation stage is enabled and the translation stage OA is larger than the implemented PA size, then an Address size fault is generated at the lookup level in the translation stage that generated the OA.

D5.1.7 Supported virtual address ranges

R_{HJYSS} The Hypervisor running at EL2 selects the memory system architecture for each Guest operating system by configuring VTCR_EL2.MSA.

I_{XGSBK} By allowing the configuration of the memory system architecture on a per guest basis allows the Hypervisor to support multiple Guest operating systems utilizing either the PMSAv8-64 or VMSAv8-64.

R_{ZBNPW} Memory accesses by both the VMSAv8-64 and PMSAv8-64 guests at EL1 are further validated by the EL2 MPU.

Note

The Secure Check Control, VSTCR_EL2.SC does not differentiate between translation table walk or memory accesses.

R_{VFFXF} If multiuple VMSAv8-64 or PMSAv8-64 guests are present, then these guests must be configured to use non-conflicting physical memory addresses.

I_{YMKBT} The VA size is determined by configuring the stage 1 translation IA size.
For more information, see Input address size configuration.

R_{XFMCS} For a translation stage that uses the VMSAv8-64 translation system, the maximun OA size that can be described by translation table entries is one of the following:

- If FEAT_LVA is not implemented, then 48 bits.
- If FEAT_LVA is implemented, then 52 bits.

I_{KRQBC} Only stage 1 address translation in the EL1&0 translation regimes support two VA ranges.

R_{BXWQW} If a translation regime supports two VA ranges, then each VA range represents an independent mapping from IA to OA.

R_{GZPNP} If a translation regime supports a single VA range, then all of the following apply:

- The maximum IA width is 56 bits.
- If the maximum VA size is 48 bits, then the maximum VA range is `0x0000_0000_0000_0000` to `0x0000_FFFF_FFFF_FFFF`.
- If the maximum VA size is 52 bits, then the maximum VA range is `0x0000_0000_0000_0000` to `0x000F_FFFF_FFFF_FFFF`.
- If the maximum VA size is 56 bits, then the maximum VA range is `0x0000_0000_0000_0000` to `0x00FF_FFFF_FFFF_FFFF`.

R_{ZNYRD} If a translation regime supports two VA ranges, then all of the following apply:

- The maximum IA width is 55 bits.
- IA bit[55] determines one of the following VA ranges:
  - If IA bit[55] is 0, the lower VA range is used.
  - If IA bit[55] is 1, the upper VA range is used.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The maximum lower VA range is one of the following:
  - If the maximum VA size is 48 bits, then the maximum VA range is `0x0000_0000_0000_0000` to `0x0000_FFFF_FFFF_FFFF`.
  - If the maximum VA size is 52 bits, then the maximum VA range is `0x0000_0000_0000_0000` to `0x000F_FFFF_FFFF_FFFF`.
- The maximum upper VA range is one of the following:
  - If the maximum VA size is 48 bits, then the maximum VA range is `0xFFFF_0000_0000_0000` to `0xFFFF_FFFF_FFFF_FFFF`.
  - If the maximum VA size is 52 bits, then the maximum VA range is `0xFFF0_0000_0000_0000` to `0xFFFF_FFFF_FFFF_FFFF`.

R_{CSSJN} If the configured size of the upper VA range is less than the maximum, then all of the following apply:

- The range starting address is greater than the address used by the maximum VA size.
- The range ending address is `0xFFFF_FFFF_FFFF_FFFF`.

I_{THWTD} A 48-bit VA range defines an address space of 256TB. A 52-bit VA range defines an address space of 4PB.

D5.1.8 Input address size configuration

R_{TLMML} If a translation stage supports a single VA range, then TCR_ELx.T0SZ configures the IA size, translated using TTBR0_ELx.

R_{GMLRF} If a translation stage supports two VA ranges, then all of the following TCR_ELx.T*n*SZ fields specify the IA size:

- TCR_ELx.T0SZ configures the IA size of the lower VA range, translated using TTBR0_ELx.
- TCR_ELx.T1SZ configures the IA size of the upper VA range, translated using TTBR1_ELx.

I_{GRZSP} The stage 1 translation IA size is defined as 64-TCR_ELx.T*n*SZ.

R_{PLCGL} For all translation stages, the following table shows how the maximum permitted T*n*SZ value is determined:

Table D5-5 Determining the maximum permitted T*n*SZ value

Translation granule size	FEAT_TTST not implemented	FEAT_TTST implemented
4KB	39	48
16KB	39	48
64KB	39	47

I_{FTBXR} If TCR_ELx.T*n*SZ configures an IA size that is smaller than the maximum size, then each one-bit reduction in the IA size has one of the following effects on the lookup level that the table walk starts with:

- The translation table size is reduced by one half.
- The table walk starts one level later.

R_{YXNYW} If the value of T*n*SZ is greater than the maximum permitted value, then any use of the T*n*SZ value causes one of the following IMPLEMENTATION DEFINED behaviors to occur:

- For all purposes other than reading back the value, the implementation behaves as if the T*n*SZ field has the maximum permitted value.
- Any use generates a level 0 Translation fault.

R_{VJFMB} For a stage 1 translation using the VMSAv8-64 translation system, the minimum Effective value of T*n*SZ is one of the following:

- For the 4KB or 16KB translation granule the minimum Effective value of T*n*SZ is 16.
- For the 64KB translation granule, one of the following:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— If FEAT_LVA is not implemented, then the minimum Effective value of T*n*SZ is 16.
— If FEAT_LVA is implemented, then the minimum Effective value of T*n*SZ is 12.

R_{VCDSW} If all of the following apply, then a stage 1 level 0 Address size fault is generated:

- Stage 1 translation is disabled.
- The IA size is larger than the implemented PA size.

I_{QZVXG} For an address translation stage, when an attempt is made to translate an address larger than the configured IA size, a level 0 Translation fault at that translation stage is generated.

R_{SXWGM} For a stage 1 address translation, if the value of T*n*SZ is smaller than the minimum Effective value, then when using the T*n*SZ value to translate an address, one of the following applies:

- If FEAT_LVA is not implemented, then it is IMPLEMENTATION DEFINED which of the following behaviors occurs:
  — For all uses other than reading back the value, the implementation behaves as if the T*n*SZ field has a value of 16.
  — When using the T*n*SZ value to translate an address, a stage 1 level 0 Translation fault is generated.
- If FEAT_LVA is implemented, then when using the T*n*SZ value to translate an address, a stage 1 level 0 Translation fault is generated.

D5.1.9 Intermediate physical address size configuration

R_{VPYYR} In the Armv8-R architecture stage 2 translations are handled by the EL2 MPU, according to the rules describing the PMSAv8-64.

I_{DDFLH} For more information, see The AArch64 Protected Memory System Architecture.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.2 Translation process

I_{GDSWV} The translation of a VA to a PA begins with reading the translation table base address register, followed by a walk through the translation tables to read the descriptors associated with the mapping.

D5.2.1 Translation table walk

R_{GXDYN} A translation table walk is the set of translation table lookups that are required to do all of the following:

- For a single stage translation at stage 1, translate a VA to a PA.
- For two translation stages and stage 1 is disabled, translate an IPA to a PA.
- For a two stage translation, all of the following:
  - For a stage 1 translation, translate a VA to an IPA.
  - For stage 2, no translation occurs and the IPA is treated as a physical address and the EL2 MPU enforces the configured access permissions.

R_{RHQPX} When a translation table walk begins, the initial translation table lookup uses the translation table base address stored in the TTBR_ELx register to locate the translation table.

R_{XFCRX} When a translation table lookup occurs, the descriptor held in the translation table entry is read as an 8-byte single-copy atomic access.
For information on the translation table alignment requirements, see Translation table alignment.

R_{YPWQG} The descriptor type indicates one of the following:

- The translation table walk is complete and the translation table entry is the final entry.
- The translation table walk requires an extra lookup at the next higher lookup level.
- The descriptor is invalid.

R_{NMTJN} All of the following descriptor types conclude the translation process:

- Page descriptor.
- Block descriptor.
- Invalid descriptor.

For more information, see Translation table descriptor formats.

I_{WMLYG} When an Invalid descriptor is returned during the translation process, a Translation fault at the current lookup level is generated.

I_{QPHCD} When an extra lookup level is required, the descriptor contains all of the following information:

- The translation table base address of the next lookup level.
- For stage 1 descriptors, if the Effective value of TCR_ELx.HPD*n* is 0, then the descriptor has hierarchical access permissions that are applied to the final translation.
- If the translation is made in a Secure translation regime, then a stage 1 descriptor indicates whether the base address is in the Secure address space or the Non-secure address space, unless a hierarchical control at a previous lookup level has indicated that it is required to be in the Non-secure address space.

For more information, see Security state of translation table lookups.

R_{VSJGV} For a translation lookup level, the translation table base address is one of the following:

- For the initial lookup level, the aligned value of the appropriate TTBR_ELx.BADDR field.
- For subsequent lookup levels, the next-level translation table base address returned by the previous lookup level.

I_{ZMMXH} When the last lookup level of a translation stage returns a valid descriptor, it contains all of the following:

- The OA from the translation stage.
- The memory region access permissions.
- The memory region attributes.

For more information, see Memory region attributes and Memory access control.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{NLLYJ} When a translation table walk succeeds, all of the following are returned:

- The OA.
- If the translation table walk is made in a Secure translation regime, then the information returned indicates whether the OA is in the Secure IPA or PA space, or the Non-secure IPA or PA space.
- The output memory region attributes.
- The output memory region access permissions.

For more information, see Security state of translation table lookups.

I_{VPFHD} The following figure is a generalized view of a single address translation stage with three lookup levels.

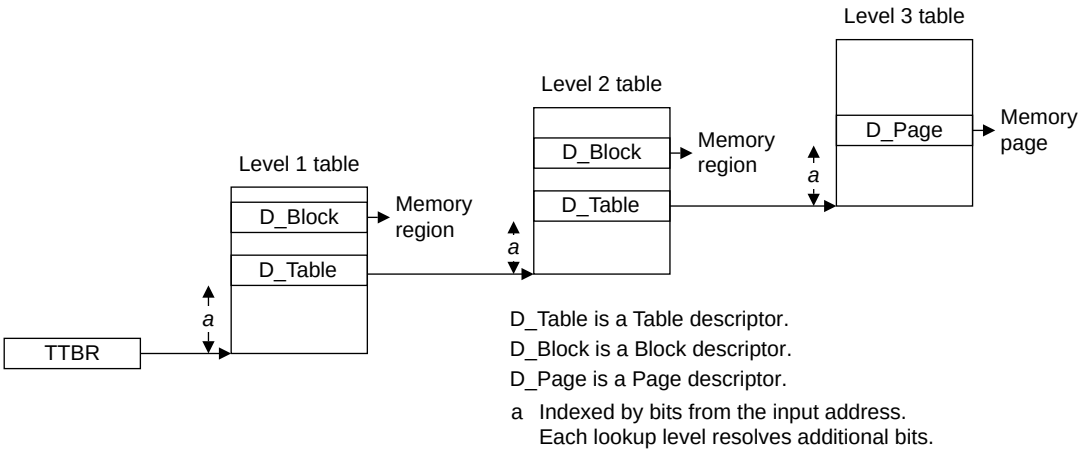


Figure D5-1 Generalized view of a single address translation stage

I_{JWQFM} If all of the following are true, then a translation table entry is permitted to be cached in a Translation Lookaside Buffer (TLB):

- The entry is valid.
- Using the entry does not generate a Translation fault, an Address size fault, or an Access flag fault.

I_{ZTRGT} TLB caching can reduce the number of required translation table lookups.

D5.2.2 Translation table base address register

I_{TLWRN} For a translation stage, the translation table base address register, TTBR_ELx, holds the translation table base address of the initial lookup.

R_{DXLKR} For a translation stage that supports two VA ranges, two translation table base address registers are required.

I_{HPFDM} Software can use TCR_ELx.TnSZ to configure the translation stage IA size to be smaller than the supported size.

R_{PNHHK} For stage 1 address translations in EL1&0, all of the following apply:

- The stage 1 translation table base address held in the TTBR_ELx register is an IPA.
- The stage 1 translation table base address returned by a Table descriptor is an IPA.
- Accesses to stage 1 translation tables are subject to a stage 2 translation.

R_{GYQHZ} For an address translation stage, the translation table base address in TTBR_ELx is defined for the supported OA size of that stage.

I_{SXSYL} For the VMSAv8-64 translation system, the number of {I}PA bits held in TTBR_ELx is determined by the granule size and OA address size. The bits used for each granule size when using the maximum OA address size of 48 bits or 52 bits

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

are shown in the following table. Software might be required to set one or more of the low-order base address bits to zero to align the table to the table size.

Table D5-6 {I}PA bits held in TTBR_ELx

Granule size	Maximum OA size	Translation table base address	{I}PA bits held in TTBR_ELx
4KB, 16KB, or 64KB	52 bits	TTBR_ELx[5:2]	{I}PA[51:48]
4KB	48 bits or 52 bits	TTBR_ELx[47:12]	{I}PA[47:12]
16KB	48 bits or 52 bits	TTBR_ELx[47:14]	{I}PA[47:14]
64KB	48 bits or 52 bits	TTBR_ELx[47:16]	{I}PA[47:16]

I_{QTPMQ} If an address translation stage uses an OA size smaller than the maximum, then the upper bits of the translation table base address in TTBR_ELx corresponding to the upper bits of OA size are required to be set by software to zero.

I_{LXTNH} If TCR_ELx.T*n*SZ specifies an IA size that is smaller than the maximum size resolved at the initial lookup level, then more low-order TTBR_ELx bits are used to hold the translation table base address.

R_{VPBBF} If an address translation stage uses an OA size smaller than the maximum and if the bits above the configured OA size of the translation table base address in TTBR_ELx are not set to zero, then an Address size fault is generated and reports all of the following:

- A translation level 0 lookup as generating the fault, regardless of whether or not the translation stage starts with a level 0 lookup.
- The translation stage that generated the fault.

R_{DYDPX} Direct writes to TTBR0_ELx and TTBR1_ELx occur in program order relative to one another, without the need for explicit synchronization. For any one translation, all indirect reads of TTBR0_ELx and TTBR1_ELx that are made as part of the translation observe only one point in that order of direct writes.

I_{RBMTM} Consistent with the general requirements for direct writes to System registers, direct writes to TTBR_ELx are not required to be observed by indirect reads until completion of a Context synchronization event.

I_{QBVNF}

Example D5-1 Changing translation table base address and ASID value when TCR_EL1.A1=0

The following is an example software sequence for changing translation table base address and ASID value when TCR_EL1.A1=0:

```
Change TTBR1 to point only at global entries
Change TTBR0 (includes changing the ASID)
Change TTBR1 to point at tables containing non-global entries
ISB
```

I_{GRBLY}

Example D5-2 Changing translation table base address and ASID value when TCR_EL1.A1=1

The following is an example software sequence for changing translation table base address and ASID value when TCR_EL1.A1=1:

```
Change TTBR0 to point to no valid entries
Change TTBR1 (includes changing the ASID)
Change TTBR0 to point at tables containing valid entries
ISB
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{BSFJP} For more information, see System registers relevant to MMU operation.

D5.2.3 Selection between TTBR0_ELx and TTBR1_ELx when two VA ranges are supported

R_{HCTPT} If a stage 1 translation regime supports two VA ranges, then the translation regime TTBR_ELx registers point to all of the following address ranges:

- For the lower VA range that begins at address `0x0000_0000_0000_0000`, TTBR0_ELx points to the translation table for the initial lookup level.
- For the upper VA range that ends at address `0xFFFF_FFFF_FFFF_FFFF`, TTBR1_ELx points to the translation table for the initial lookup level.

R_{YYVYV} If a stage 1 translation regime supports two VA ranges, then the TCR_ELx.{T0SZ, T1SZ} fields configure all of the following address range sizes:

- The lower VA range is `0x0000_0000_0000_0000` to $(2^{(64-T0SZ)} - 1)$.
- The upper VA range is $(2^{64} - 2^{(64-T1SZ)})$ to `0xFFFF_FFFF_FFFF_FFFF`.

I_{THCZK} If a stage 1 translation regime supports two VA ranges, when an accessed address is not in the lower VA range or the upper VA range, a stage 1 level 0 Translation fault is generated.

I_{BCMMD} The following figure illustrates the two address ranges translated by the tables the TTBR_ELx registers point to.

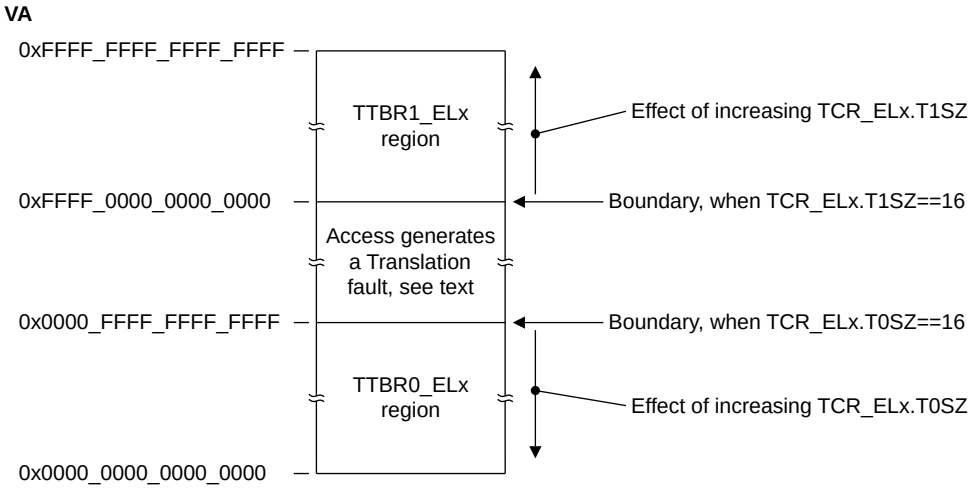


Figure D5-2 Example of a stage 1 translation that supports two VA ranges

R_{VZCSR} If a stage 1 translation regime supports two VA ranges, then all of the following are used to select the TTBR_ELx:

- If VA bit[55] is zero, then TTBR0_ELx is selected.
- If VA bit[55] is one, then TTBR1_ELx is selected.

I_{ZFSYQ} If a stage 1 translation regime supports two VA ranges and TCR_ELx.EPD*n* is one, then when a TLB miss occurs based on TTBRn_ELx, a level 0 Translation fault is generated, and no translation table walk is done.

I_{QBLKT} For more information, see Address tagging.

D5.2.3.1 Preventing EL0 access to halves of the address map

R_{BNDVG} All statements in this section require implementation of FEAT_E0PD.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{PZZHN} The TCR_ELx.{E0PD0, E0PD1} fields can be used to prevent EL0 access to the addresses translated by the corresponding TTBR0_ELx or TTBR1_ELx.

I_{RFRSQ} When the TCR_ELx.{E0PD0, E0PD1} fields prevent EL0 access to an address translated by TTBR0_ELx or TTBR1_ELx, then a level 0 Translation fault is generated.

R_{DDFWH} When the TCR_ELx.{E0PD0, E0PD1} fields generate a level 0 Translation fault, then all of the following apply:

- The time needed to take the fault should be the same whether or not the address accessed is present in a TLB, to mitigate attacks that use fault timing.
- The fault generated does not affect any micro-architectural state of the PE in a manner that is different if the address accessed is present in a TLB or not, to prevent this information being used to determine the presence of the address in a TLB.

R_{CZWGX} When the TCR_ELx.{E0PD0, E0PD1} fields generate a level 0 Translation fault, the fault is not counted as a TLB miss for performance monitoring features.

D5.2.4 Translation table and translation table lookup properties

I_{SLRTY} Translation table and translation table lookup properties include the table size, table alignment, table endianness, and memory attributes.

D5.2.4.1 Translation table size

R_{RVHGW} The descriptor size in a translation table entry is an eight-byte, or 64-bit, object.

R_{WDRNX} If *n* is the number of bits resolved by a lookup level, then the number of translation table entries required at that lookup level is 2^n.

R_{NVTHS} The size of a translation table in bytes is determined by multiplying the number of entries by the descriptor size.

I_{KNSPW} The maximum number of translation table entries is determined by the translation granule size, which is defined by one of the following:

- For a stage 1 translation that supports one VA range, TCR_ELx.TG0.
- For a stage 1 translation that can support two VA ranges, all of the following:
  - For the lower VA range, TCR_ELx.TG0.
  - For the upper VA range, TCR_ELx.TG1.

D5.2.4.2 Translation table alignment

R_{KBLCR} A translation table is required to be aligned to one of the following:

- For the VMSAv8-64 translation system, if the translation table has fewer than eight entries and an OA size greater than 48 bits is used, then the table is aligned to 64 bytes.
- Otherwise, the translation table is aligned to the size of that translation table.

I_{TBPFK} Only when all of the following are true is it possible to have fewer than 8 translation table entries:

- The translation table is at the initial lookup level.
- Concatenated translation tables are not used.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.2.4.3 Translation table lookup endianness

I_{KLNSQ} When a translation table lookup reads a translation table entry the read is an 8-byte single-copy atomic access.

R_{FDKMS} The endianness of a translation table lookup is determined by the appropriate SCTLR_EL1, System Control Register (EL1).EE bit.

R_{PZVGH} Changing an SCTLR_EL1, System Control Register (EL1).EE bit requires synchronization before the change is guaranteed to be visible to subsequent operations.
For more information, see Synchronization requirements for AArch64 System registers.

I_{QJDJV} Arm does not recommend changing an SCTLR_EL1, System Control Register (EL1).EE bit unless one or more of the following apply:

- The translation table lookups affected by the modified SCTLR_EL1, System Control Register (EL1).EE bit belong to an out-of-context translation regime.
- For any translation stage affected by the modified SCTLR_EL1, System Control Register (EL1).EE bit, address translation is disabled.

D5.2.4.4 Translation table lookup memory attributes

I_{TXCPH} For a translation table lookup in an address translation stage, the TCR_ELx and VTCR_EL2 registers determine the memory Cacheability and Shareability attributes that apply.

R_{ZZSYD} For a translation table lookup in an address translation stage, the required memory type is Normal memory.

R_{XGXKR} When memory locations that hold the translation tables are accessed by software, and the memory accesses and the translation table walks have mismatched memory attributes, the behavior is the same as different memory accesses to the same memory location using mismatched memory attributes.
For more information, see Mismatched memory attributes.

I_{BSKKL} Arm recommends that the memory attributes applied by the TCR_ELx to the translation tables are the same as the attributes that are applied by other accesses to the memory that holds the translation tables.

I_{WVTXS} For more information on Normal and Device memory and on the Cacheability and Shareability attributes, see Memory types and attributes.

D5.2.5 Translation table walk properties

D5.2.5.1 Ordering of memory accesses from translation table walks

R_{TGRMW} A translation table walk is considered to be a separate observer. An explicit write to the translation tables might be observed by that separate observer for either of the following:

- A translation table walk caused by a different explicit write generated by the same instruction.
- A translation table walk caused by an explicit access generated by any instruction appearing in program order after the instruction doing the explicit write to the translation table.

R_{SPVBD} The explicit write to the translation tables is guaranteed to be observable, to the extent required by the Shareability attributes, only after the execution of a DSB instruction. This DSB instruction and the instruction that performed the explicit write to the translation tables must have been executed by the same PE.

R_{RGPMV} Any writes to the translation tables are not observed by any translation table accesses associated with an explicit memory access generated by a load or store that occurs in program order before the instruction that performs the write to the translation tables.

I_{LTSNZ} For more information, see:

- External visibility requirement.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Memory barriers.

D5.2.5.2 Security state of translation table lookups

R_{PXRPP} For a translation table walk in a Non-secure translation regime, all translation table lookups are done to Non-secure IPA or PA space.

I_{KGBDL} The OA space of the Secure EL1&0 stage 1 translation regime is PA space.

R_{SCHHQ} For a stage 1 translation table walk in a Secure Translation Regime, the OA space at stage 1 of the translation table lookups is determined by all of the following:

- The initial translation table lookup is done to Secure OA space.
- When the Table descriptor lookup is to Secure OA space, the descriptor NSTable bit determines one of the following:
  - If NSTable is zero, the next translation table lookup is done to Secure OA space.
  - If NSTable is one, the next translation table lookup is done to Non-secure OA space.
- When the Table descriptor lookup is to Non-secure OA space, the next translation table lookup is done to Non-secure OA space.

For more information, see Hierarchical control of Secure or Non-secure memory accesses.

D5.2.6 VMSAv8-64 translation using the 4KB granule

R_{LCVYF} All statements in this section and subsections require a translation stage use the VMSAv8-64 translation system.

I_{SZLLC} Address translations that use the 4KB granule have a 4KB page size. Depending on the settings and supported features, up to 40 address bits are resolved using up to 5 lookup levels.

R_{NNSVF} For the 4KB translation granule, OA[51:48] are treated as `0b0000`.

R_{SMKJL} For each lookup level supported by the 4KB translation granule, the following table describes the translation table properties at that level.

Table D5-7 4KB granule translation table properties at each lookup level

Lookup level	Index into translation table	Maximum entries in table	Contents of translation table entries
-1	-	-	Lookup level not supported
0	IA[47:39]	512	Table descriptors
1	IA[38:30]	512	Table descriptors and Block descriptors
2	IA[29:21]	512	Table descriptors and Block descriptors
3	IA[20:12]	512	Page descriptors

R_{VHWNS} For the 4KB translation granule, a translation resolved by a Page descriptor at lookup level 3 has all of the following properties:

- The page size is 4KB.
- The translation can resolve a page using IA[47:12].
- The page is addressed by OA[47:12].
- IA[11:0] is mapped directly to OA[11:0].

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{NHKBC} For the 4KB translation granule, a translation resolved by a Block descriptor at the specified lookup level has all of the properties shown in the following table:

Table D5-8 4KB granule block descriptor properties at each lookup level

Lookup level	Size of memory region addressed by Block descriptor	Bit range that is direct mapped from IA to OA	IA bit range that selects Block descriptor
0	Not supported	-	-
1	1GB	IA[29:0] maps to OA[29:0]	IA[47:30]
2	2MB	IA[20:0] maps to OA[20:0]	IA[47:21]

D5.2.6.1 VMSAv8-64 Stage 1 address translation using the 4KB translation granule

R_{GWNBR} For the 4KB translation granule, when a stage 1 translation table walk is started, the initial lookup level is determined by the value of the TCR_ELx.T*n*SZ field as shown in the following table.

Table D5-9 4KB granule, determining stage 1 initial lookup level

Initial lookup level	T*n*SZ minimum value	Maximum IA bits resolved	T*n*SZ maximum value	Minimum IA bits resolved	Additional requirements
0	16	IA[47:12]	24	IA[39:12]	-
1	25	IA[38:12]	33	IA[30:12]	-
2	34	IA[29:12]	39	IA[24:12]	-
2	40	IA[23:12]	42	IA[21:12]	FEAT_TTST is implemented
3	43	IA[20:12]	48	IA[15:12]	FEAT_TTST is implemented

I_{JCHLN} For a stage 1 translation in the 4KB translation granule, depending on the IA size, the initial lookup level is indexed by up to 9 bits and all remaining lookup levels are indexed by exactly 9 bits.

D5.2.6.2 Finding the descriptor when using the VMSAv8-64 4KB translation granule

R_{VMPRR} For the 4KB translation granule, the following table shows the algorithm to find the descriptor address at each supported lookup level, using of all of the following information:

- The translation table base address, *BaseAddr*.
- The number of bits in the supported PA size, *PAsize*.
- The IA supplied to the translation stage and used as an index into the translation table.
- For each initial lookup level, the permitted range of values for TCR_ELx.T*n*SZ.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D5-10 4KB granule, finding the descriptor address

Lookup level	Stage 1 translation table descriptor address
-1	if 12 <= T*n*SZ <= 15 then *x* = (19 - T*n*SZ) BaseAddr[PAsize-1:*x*]:IA[*x*+44:48]:`0b000`
0	if 16 <= T*n*SZ <= 24 then *x* = (28 - T*n*SZ) else *x* = 12 BaseAddr[PAsize-1:*x*]:IA[*x*+35:39]:`0b000`
1	if 25 <= T*n*SZ <= 33 then *x* = (37 - T*n*SZ) else *x* = 12 BaseAddr[PAsize-1:*x*]:IA[*x*+26:30]:`0b000`
2	if 34 <= T*n*SZ <= 42 else *x* = 12 BaseAddr[PAsize-1:*x*]:IA[*x*+17:21]:`0b000`
3	if 43 <= T*n*SZ <= 48 then *x* = (55 - T*n*SZ) else *x* = 12 BaseAddr[PAsize-1:*x*]:IA[*x*+8:12]:`0b000`

D5.2.7 VMSAv8-64 translation using the 16KB granule

R_{QSQNQ} All statements in this section and subsections require a translation stage use the VMSAv8-64 translation system.

I_{TDTCY} Address translations that use the 16KB granule have a 16KB page size. Depending on the settings and supported features, up to 38 address bits are resolved using up to 4 lookup levels.

I_{JCYXY} For the 16KB translation granule, the maximum VA and PA supported by a translation regime is 48 bits.

R_{CDLYH} For the 16KB translation granule then OA[51:48] are treated as `0b0000`.

R_{CWKJW} For each lookup level supported by the 16KB translation granule, the following table describes the translation table properties at that level.

Table D5-11 16KB granule translation table properties at each lookup level

Lookup level	Index into translation table	Maximum entries in table	Contents of translation table entries
0	IA[47]	232	Table descriptors Lookup level 0 supported only at stage 1
1	IA[46:36]	2048	Table descriptors
2	IA[35:25]	2048	Table descriptors and Block descriptors
3	IA[24:14]	2048	Page descriptors

R_{YHJLX} For the 16KB translation granule, a translation resolved by a Page descriptor at lookup level 3 has all of the following properties:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The page size is 16KB.
- The translation can resolve a page using the maximum address ranges IA[47:14].
- The page is addressed by OA[47:14].
- IA[13:0] is mapped directly to OA[13:0].

R_{FFHSQ} For the 16KB translation granule, a translation resolved by a Block descriptor at the specified lookup level has all of the properties shown in the following table:

Table D5-12 16KB granule block descriptor properties at each lookup level

Lookup level	Size of memory region addressed by Block descriptor	Bit range that is direct mapped from IA to OA	IA bit range that selects Block descriptor
1	Not supported	-	-
2	32MB	IA[24:0] maps to OA[24:0]	IA[47:25]

D5.2.7.1 VMSAv8-64 Stage 1 address translation using the 16KB translation granule

R_{WDCVZ} For the 16KB translation granule, when a stage 1 translation table walk is started, the initial lookup level is determined by the value of the TCR_ELx.T*n*SZ field as shown in the following table:

Table D5-13 16KB granule, determining stage 1 initial lookup level

Initial lookup level	T*n*SZ minimum value	Maximum IA bits resolved	T*n*SZ maximum value	Minimum IA bits resolved	Additional requirements
0	16	IA[47:14]	16	IA[47:14]	-
1	17	IA[46:14]	27	IA[36:14]	-
2	28	IA[35:14]	38	IA[25:14]	-
3	39	IA[24:14]	39	IA[24:14]	-
3	40	IA[23:14]	48	IA[15:14]	FEAT_TTST is implemented

I_{FLDKN} For a stage 1 translation in the 16KB translation granule, depending on the IA size, the initial lookup level is indexed by up to 11 bits and all remaining lookup levels are indexed by exactly 11 bits.

I_{KSSHJ} For the 16KB translation granule and a 52-bit IA, the following figure is a generalized view of a stage 1 address translation. An IA and an OA larger than 48 bits requires that the Effective value of TCR_ELx.DS is 1. The 64GB block size requires that the Effective value of TCR_ELx.DS is 1.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

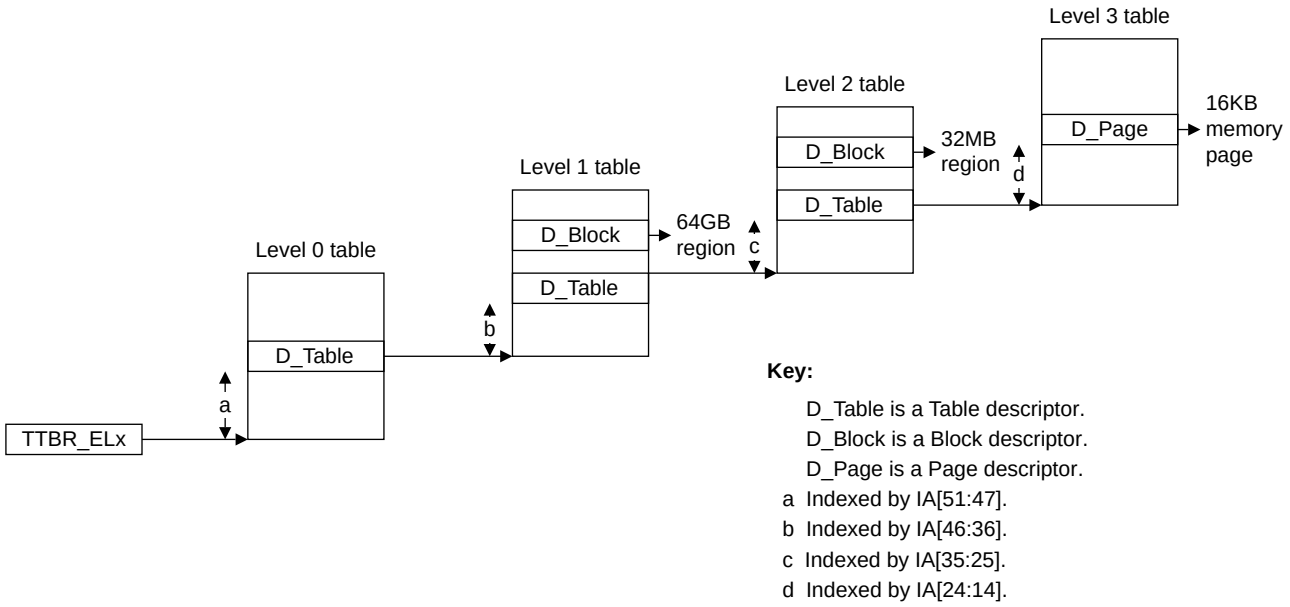


Figure D5-3 Generalized view of a stage 1 address translation using the 16KB granule

D5.2.7.2 Finding the descriptor when using the VMSAv8-64 16KB translation granule

R_{VVQFC} For the 16KB translation granule, the following table shows the algorithm to find the descriptor address at each supported lookup level, using of all of the following information:

- The translation table base address, *BaseAddr*.
- The number of bits in the supported PA size, *PAsize*.
- The IA supplied to the translation stage and used as an index into the translation table.
- For each initial lookup level, the permitted range of values for TCR_ELx.T*n*SZ.

Table D5-14 16KB granule, finding the descriptor address

Lookup level	Stage 1 translation table descriptor address
0	if 12 <= T*n*SZ <= 16 then *x* = (20 - T*n*SZ) BaseAddr[PAsize-1:*x*]:IA[*x*+43:47]:`0b000`
1	if 17 <= T*n*SZ <= 27 then *x* = (31 - T*n*SZ) else *x* = 14 BaseAddr[PAsize-1:*x*]:IA[*x*+32:36]:`0b000`
2	if 28 <= T*n*SZ <= 38 then *x* = (42 - T*n*SZ) else *x* = 14 BaseAddr[PAsize-1:*x*]:IA[*x*+21:25]:`0b000`
3	if 39 <= T*n*SZ <= 48 then *x* = (53 - T*n*SZ) else *x* = 14 BaseAddr[PAsize-1:*x*]:IA[*x*+10:14]:`0b000`

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.2.8 VMSAv8-64 translation using the 64KB granule

R_{RGNZV} All statements in this section and subsections require a translation stage use the VMSAv8-64 translation system.

I_{VDPWL} Address translations that use the 64KB granule have a 64KB page size. Depending on the settings and supported features, up to 36 address bits are resolved using up to 3 lookup levels.

I_{LRFQD} For the 64KB translation granule, the maximum VA size supported by a translation regime is one of the following:

- If FEAT_LVA is not implemented, then the maximum VA supported is 48 bits.
- If FEAT_LVA is implemented, then the maximum VA supported is 52 bits.

I_{YZDPC} For the 64KB translation granule, the maximum PA size supported by a translation regime is one of the following:

- If FEAT_LPA is not implemented, then the maximum PA supported is 48 bits.
- If FEAT_LPA is implemented, then the maximum PA supported is 52 bits.

R_{NSQXP} For the 64KB translation granule, if FEAT_LPA is not implemented, then OA[51:48] are treated as `0b0000`.

R_{MLLGN} For each lookup level supported by the 64KB translation granule, the following table describes the translation table properties at that level.

Table D5-15 64KB granule translation table properties at each lookup level

Lookup level	Index into translation table	Maximum entries in table	Contents of translation table entries	Additional requirements for stage 1
1	IA[47:42]	64	Table descriptors	-
			Table descriptors and Block descriptors	FEAT_LPA
	IA[51:42]	1024	Table descriptors	FEAT_LVA
			Table descriptors and Block descriptors	FEAT_LVA, FEAT_LPA
2	IA[41:29]	8192	Table descriptors and Block descriptors	-
3	IA[28:16]	8192	Page descriptors	-

R_{RVYWB} For the 64KB translation granule, a translation resolved by a Page descriptor at lookup level 3 has all of the following properties:

- The page size is 64KB.
- For stage 1, the translation can resolve a page using one of the following maximum address ranges:
  - If FEAT_LVA is not implemented, then IA[47:16].
  - If FEAT_LVA is implemented, then IA[51:16].
- For stage 1, the page is addressed by one of the following:
  - If FEAT_LPA is not implemented, then OA[47:16].
  - If FEAT_LPA is implemented, then OA[51:16].
- IA[15:0] is mapped directly to OA[15:0].

R_{RBNGS} For the 64KB translation granule, a translation resolved by a Block descriptor at the specified lookup level has all of the properties shown in the following table:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D5-16 64KB granule block descriptor properties at each lookup level

Lookup level	Size of memory region addressed by Block descriptor	Bit range that is direct mapped from IA to OA	IA bit range that selects Block descriptor	Additional requirements for stage 1	Additional requirements for stage 2
1	4TB	IA[41:0] maps to OA[41:0]	IA[47:42]	FEAT_LPA	FEAT_LPA
			IA[51:42]	FEAT_LVA, FEAT_LPA	FEAT_LPA
2	512MB	IA[28:0] maps to OA[28:0]	IA[47:29]	-	-
			IA[51:29]	FEAT_LVA	FEAT_LPA

I_{TVDSK} For the 64KB translation granule, the following figure shows how the maximum 52-bit IA size is resolved. For a stage 1 translation, an IA larger than 48 bits requires the implementation of FEAT_LVA.

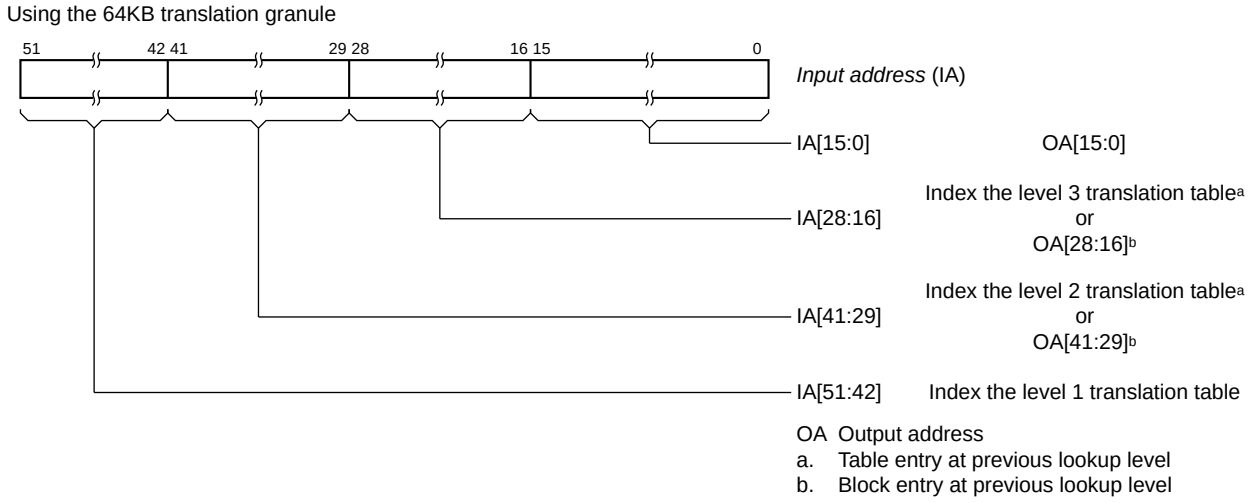


Figure D5-4 52-bit IA resolved using 64KB translation granule

D5.2.8.1 VMSAv8-64 Stage 1 address translation using the 64KB translation granule

R_{FMBKV} For the 64KB translation granule, when a stage 1 translation table walk is started, the initial lookup level is determined by the value of the TCR_ELx.T*n*SZ field as shown in the following table:

Table D5-17 64KB granule, determining stage 1 initial lookup level

Initial lookup level	T*n*SZ minimum value	Maximum IA bits resolved	T*n*SZ maximum value	Minimum IA bits resolved	Additional requirements
1	12	IA[51:16]	15	IA[48:16]	FEAT_LVA is implemented
1	16	IA[47:16]	21	IA[42:16]	-
2	22	IA[41:16]	34	IA[29:16]	-
3	35	IA[28:16]	39	IA[24:16]	-
3	40	IA[23:16]	47	IA[16:16]	FEAT_TTST is implemented

I_{FQFFW} For a stage 1 translation in the 64KB translation granule, depending on the IA size, the initial lookup level is indexed by up to 13 bits and all remaining lookup levels are indexed by exactly 13 bits.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{MRDWZ} For the 64KB translation granule and a 52-bit IA, the following figure is a generalized view of a stage 1 address translation. An IA larger than 48 bits requires implementation of FEAT_LVA, and an OA larger than 48 bits requires implementation of FEAT_LPA. The 4TB block size requires implementation of FEAT_LPA.

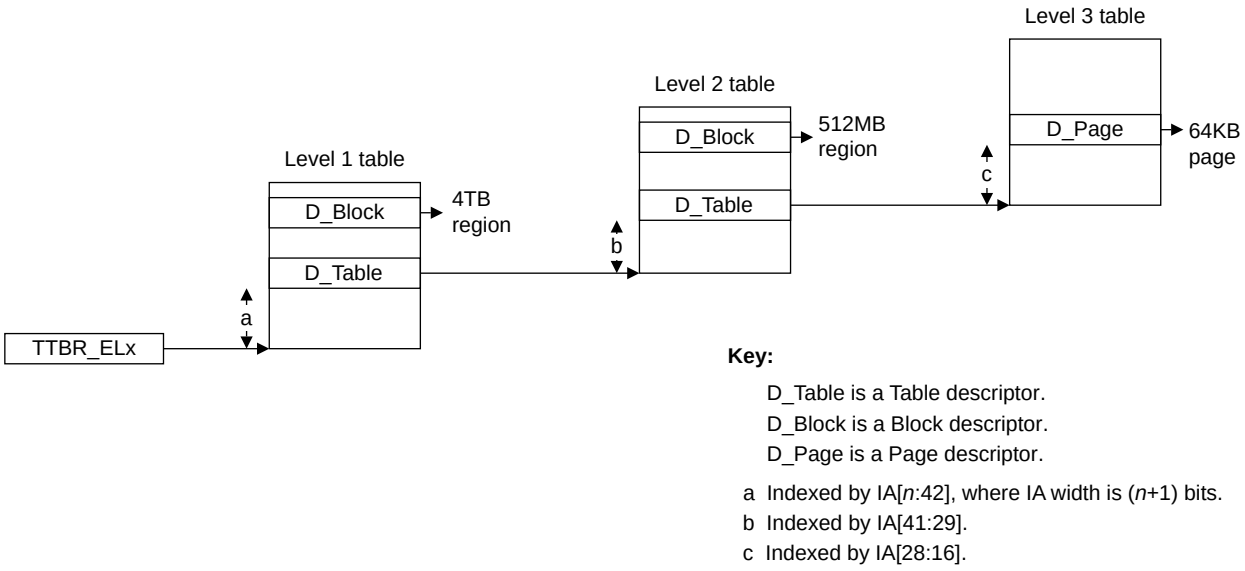


Figure D5-5 Generalized view of a stage 1 address translation using the 64KB granule

D5.2.8.2 Finding the descriptor when using the VMSAv8-64 64KB translation granule

R_{VYNTC} For the 64KB translation granule, the following table shows the algorithm to find the descriptor address at each supported lookup level, using of all of the following information:

- The translation table base address, *BaseAddr*.
- The number of bits in the supported PA size, *PAsize*.
- The IA supplied to the translation stage and used as an index into the translation table.
- For each initial lookup level, the permitted range of values for TCR_ELx.T*n*SZ.

Table D5-18 64KB granule, finding the descriptor address

Lookup level	Stage 1 translation table descriptor address
1	if 12 <= T*n*SZ <= 21 then *x* = (25 - T*n*SZ) BaseAddr[PAsize-1:*x*]:IA[*x*+38:42]:`0b000`
2	if 22 <= T*n*SZ <= 34 then *x* = (38 - T*n*SZ) else *x* = 16 BaseAddr[PAsize-1:*x*]:IA[*x*+25:29]:`0b000`
3	if 35 <= T*n*SZ <= 47 then *x* = (51 - T*n*SZ) else *x* = 16 BaseAddr[PAsize-1:*x*]:IA[*x*+12:16]:`0b000`

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.2.9 The effects of disabling an address translation stage

I_{QMGZS} Stage 1 tranlsations can be disabled. Setting VTCR_EL2.MSA to 0 disables the VMSAv8-64 memory architecture and enables the PMSAv8-64 memory architecture.

For more information, see The AArch64 Protected Memory System Architecture

D5.2.9.1 Effect of disabling address translation on maintenance and address translation instructions

R_{SJBMV} Cache maintenance instructions act on the target location regardless of all of the following:

- Whether or not any address translation stages are disabled.
- The memory attribute values.

R_{ZSSPZ} For an address translation stage that is disabled, cache maintenance instructions use flat address mapping.

R_{FZYGN} If an address translation stage is disabled, TLB maintenance operations are not affected.

I_{HRNPD} For more information, see:

- A64 Cache maintenance instructions.
- Address translation instructions.
- TLB maintenance instructions.

D5.2.10 Address translation instructions

I_{TMFCK} Address translation instructions return the result of translating an IA using a specified translation stage or regime.

I_{NQTNR} An address translation instruction has all of the following properties:

- An IA is supplied as the argument to the instruction.
- The instruction encoding determines the translation stage and regime used by the translation.
- The PAR_EL1 register is updated with the translation result.
- If an address translation stage is controlled by a higher Exception level than the Exception level at which the address translation instruction is executed, then the instruction is UNDEFINED.

I_{TSJPJ} The A64 assembly language syntax of an address translation instruction is `AT <operation>, <Xt>`.

I_{RTMLK} The `<operation>` in `AT <operation>, <Xt>` is one of the following:

- `S1E1R`, `S1E1W`, `S1E0R`, `S1E0W`, `S12E1R`, `S12E1W`, `S12E0R`, `S12E0W`, `S1E2R`, `S1E2W`, `S1E3R`, or `S1E3W`.
- If FEAT_PAN2 is implemented, `then S1E1WP` or `S1E1RP`.

I_{SLTTL} The `<operation>` in `AT <operation>, <Xt>` has a structure of `<stages><el>(<read|write>{<pan>})|<ignore>` with all of the following components:

- `<stages>` is one of the following address translation stages:
  - `S1` specifies a stage 1 translation.
- `<el>` is one of the following Exception levels that apply to the translation:
  - `E0` specifies EL0.
  - `E1` specifies EL1.
  - `E2` specifies EL2.
- `<read|write>` is one of the following:
  - `R` specifies Read.
  - `W` specifies Write.
- If FEAT_PAN2 is implemented, then the `<pan>` component is included and has all of the following properties:
  - `P` determines whether the PSTATE.PAN value is considered when determining permissions. For more information, see PSTATE.PAN.
  - The field is optional.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— If `<stages>=S1` and `<el>=E1`, the field is permitted.
— If `<stages>!=S1` or `<el>!=E1`, the field is not permitted.

I_{XLYWF} If `<el>` is higher than the current Exception level, then the address translation instruction is UNDEFINED.

I_{SYHWD} The `<Xt>` in `AT <operation>, <Xt>` specifies the 64-bit register holding the address to be translated.

R_{NJGLC} When an address is translated by an address translation instruction, no alignment restrictions exist and therefore an address translation instruction cannot generate an Alignment fault.

I_{LRNWF} Execution of an AT instruction may result in a successful translation reported in PAR_EL1, an MMU fault being reported in PAR_EL1, or an MMU fault generating an exception.
For more information see:

- MMU faults generated by address translation instructions
- Address translation instructions, successful address translation

R_{NWDPJ} When an address translation instruction is executed, the specified translation stage and regime determines all of the following:

- The entries in TLB caching structures that are permitted to be used.
- How the translation table walk is done.

R_{NPZWX} When an address translation instruction is executed, it is IMPLEMENTATION DEFINED whether the result can be returned from a TLB, or a translation table walk occurs.

I_{WGXJK} If TLB entries might differ from the underlying translation tables held in memory, such as when waiting for a maintenance or synchronization event to complete after updating the translation tables, Arm recommends not using the address translation instructions.

D5.2.10.1 Address translation instructions, successful address translation

R_{LRYRY} When an address translation instruction successfully translates an address, all of the following are updated in PAR_EL1:

- PAR_EL1.F is set to 0.
- The resulting OA is returned in PAR_EL1.PA, and the resulting attributes are returned in the other fields of PAR_EL1.

I_{PVCPF} When populating PAR_EL1 with the result of an address translation instruction, the fetches of stage 1 or stage 2 descriptors use the appropriate MECID register values for the translation regime.

I_{QKWHX} The architecture defines a single PAR, PAR_EL1, that is used regardless of all of the following:

- The Exception level at which the address translation instruction was executed.
- The Exception level that controls the address translation stage or stages used by the address translation instruction.

D5.2.10.2 Address translation instructions, effect of translation regime

I_{KJDTT} The value of HCR_EL2.DC affects all of the following instructions:

- `AT S1E0*`.
- `AT S1E1*`.
- `AT S12E0*`.
- `AT S12E1*`.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.2.10.3 Address translation instructions, synchronization requirements

I_{JTYVP} When an address translation instruction is executed, explicit synchronization is required to guarantee the result is visible to subsequent direct reads of PAR_EL1.

For more information, see Synchronization requirements for AArch64 System registers.

I_{PCPWC} Requiring explicit synchronization after executing an address translation instruction is consistent with the general requirement that the effect of a write to a System register requires explicit synchronization before the result is guaranteed to be visible to subsequent instructions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.3 Translation table descriptor formats

R_{KSQZW} If an address translation stage is controlled by an Exception level that is using AArch64, then the translation table uses 64-bit descriptors.

R_{JXLYN} A translation descriptor has one of the following formats:

- An invalid descriptor format.
- A Table descriptor format that points to the next-level translation table.
- A Block descriptor or Page descriptor format that defines the memory access properties.
- A reserved format.

R_{KHVQT} The value of descriptor bit[0] determines one of the following:

- If descriptor bit[0] is 0, then the descriptor is invalid.
- If descriptor bit[0] is 1, then the descriptor is valid.

R_{RWMFF} For 64-bit descriptors used by the VMSAv8-64 translation system, if descriptor bit[0] is 1, then the descriptor type is determined by one of the following values of descriptor bit[1]:

- For lookup levels less than lookup level 3, one of the following:
  - If bit[1] is 0, then the descriptor is a Block descriptor.
  - If bit[1] is 1, then the descriptor is a Table descriptor.
- For lookup level 3, one of the following:
  - If bit[1] is 0, then the descriptor is reserved, and treated as invalid.
  - If bit[1] is 1, then the descriptor is a Page descriptor.

Table D5-19 Determination of descriptor type, VMSAv8-64 translation system

Descriptor bit[0]	bit[1]	Descriptor type	Condition
0	-	Invalid	-
1	0	Block	Lookup level is not 3
		Reserved, treated as invalid	Lookup level is 3
	1	Table	Lookup level is not 3
		Page	Lookup level is 3

R_{CQGPJ} For Table descriptors, if the Effective value of the SKL field is 0, then the next-level table address is one of the following:

- For a level -2 Table descriptor, the base address of a level -1 table.
- For a level -1 Table descriptor, the base address of a level 0 table.
- For a level 0 Table descriptor, the base address of a level 1 table.
- For a level 1 Table descriptor, the base address of a level 2 table.
- For a level 2 Table descriptor, the base address of a level 3 table.

D5.3.1 VMSAv8-64 descriptor formats

D5.3.1.1 VMSAv8-64 Table descriptor format

I_{TWKLX} The following figure shows all of following stage 1 VMSAv8-64 Table descriptor formats:

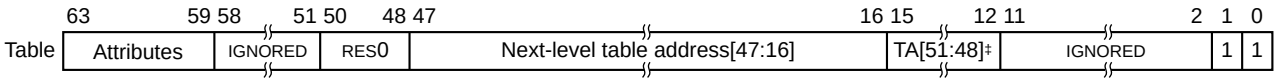
- 4KB, 16KB, and 64KB granules using a 48-bit next-level table address.
- If FEAT_LPA is implemented, the 64KB granule using a 52-bit next-level table address.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

4KB and 16KB granules, 52-bit OA

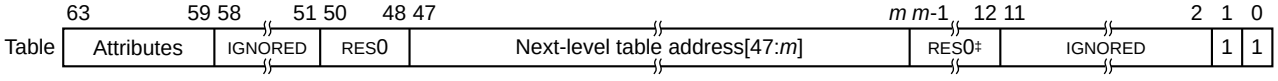
With the 4KB granule size m is 12‡ and with the 16KB granule size m is 14.
‡ When *m* is 12, the RES0 field shown for bits[(*m*-1):12] is absent.

64KB granule, 52-bit OA



‡ TA[51:48] indicates bits [51:48] of the next-level table address.

4KB, 16KB, and 64KB granules, 48-bit OA



With the 4KB granule size m is 12‡, with the 16KB granule size m is 14, and with the 64KB granule size m is 16.
‡ When *m* is 12, the RES0 field shown for bits[(*m*-1):12] is absent.

Figure D5-6 VMSAv8-64 Table descriptor formats

I$_{LZSBS}$ For a stage 1 translation, the following figure shows the next-level attribute fields in a VMSAv8-64 Table descriptor.

- Bits[58:51] are IGNORED.

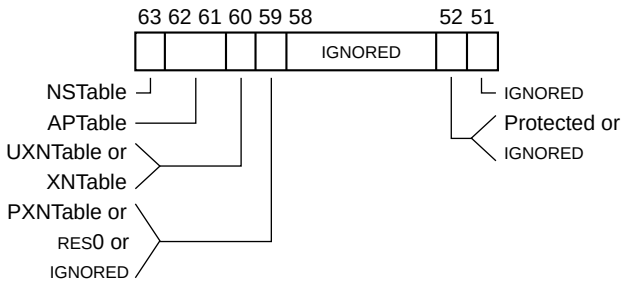


Figure D5-7 Stage 1 next-level attribute fields in a VMSAv8-64 Table descriptor

R$_{CGNCT}$ For stage 1 translations, the following table defines the fields in the VMSAv8-64 Table descriptor. In this table, NLTA is the next level table address.

Table D5-20 Stage 1 VMSAv8-64 Table descriptor fields

Bit position	Field	Condition
[63]	NSTable	Secure state. See Hierarchical control of Secure or Non-secure memory accesses.
[62:61]	APTable[1:0]	Hierarchical permissions are enabled. See Hierarchical control of data access Direct permissions.
[60]	XNTable	Hierarchical permissions are enabled and the translation regime supports a single privilege level. See Hierarchical control of instruction execution for Direct permissions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bit position	Field	Condition
	UXNTable	Hierarchical permissions are enabled and the translation regime supports two privilege levels. See Hierarchical control of instruction execution for Direct permissions.
[59]	PXNTable	Hierarchical permissions are enabled and the translation regime supports two privilege levels. See Hierarchical control of instruction execution for Direct permissions.
[58:53]	IGNORED	-
[52]	IGNORED	The Effective value of PnCH is 0.
	Protected	The Effective value of PnCH is 1. See Stage 1 Protected Attribute.
[51]	IGNORED	-
[50]	RES0	-
[49:48]	RES0	The 4KB or 16KB or the 64KB translation granule is used.
[47:12]	NLTA[47:12]	The 4KB translation granule is used.
	NLTA[47:14]	The 16KB translation granule is used. Descriptor bits [13:12] are RES0.
	NLTA[47:16]	The 64KB translation granule is used and FEAT_LPA is not implemented. Descriptor bits [15:12] are RES0.
	NLTA[47:16]: NLTA[51:48]	The 64KB translation granule is used and FEAT_LPA is implemented.
[11]	IGNORED	-
[10]	IGNORED	Hardware managed Table descriptor Access flag is not enabled.
	Access flag (AF)	Hardware managed Table descriptor Access flag is enabled. See Hardware management of the Table descriptor Access Flag.
[9:8]	IGNORED	The 4KB or 16KB or the 64KB translation granule is used.
[7:2]	IGNORED	-
[1]	Table descriptor	1, for lookup levels less than lookup level 3.
[0]	Valid descriptor	1.

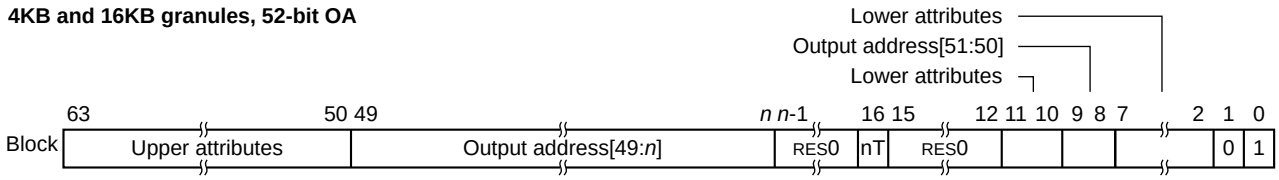
D5.3.1.2 VMSAv8-64 Block descriptor and Page descriptor formats

I_{DKGVG} The following figure shows all of the following VMSAv8-64 Block descriptor formats:

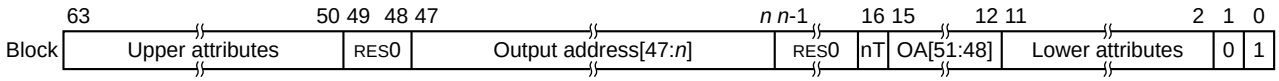
- 4KB, 16KB, and 64KB granules using a 48-bit OA.
- If FEAT_LPA is supported, the 64KB granule using a 52-bit OA.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

4KB and 16KB granules, 52-bit OA



64KB granule, 52-bit OA

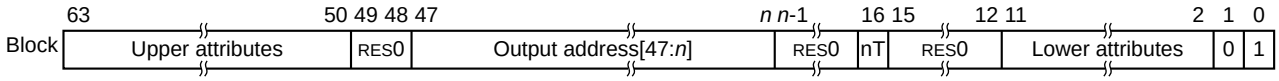


For the 4KB granule size, the level 0 descriptor *n* is 39, the level 1 descriptor *n* is 30, and the level 2 descriptor *n* is 21.

For the 16KB granule size, the level 1 descriptor *n* is 36, and the level 2 descriptor *n* is 25.

For the 64KB granule size, the level 1 descriptor *n* is 42, and the level 2 descriptor *n* is 29.

4KB, 16KB, and 64KB granules, 48-bit OA



For the 4KB granule size, the level 1 descriptor *n* is 30, and the level 2 descriptor *n* is 21.

For the 16KB granule size, the level 2 descriptor *n* is 25.

For the 64KB granule size, the level 2 descriptor *n* is 29.

Figure D5-8 VMSAv8-64 Block descriptor formats

I_{WNCLB} The following figure shows all of the following VMSAv8-64 Page descriptor formats:

- 4KB, 16KB, and 64KB granules using a 48-bit OA.
- If FEAT_LPA is supported, the 64KB granule using a 52-bit OA.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

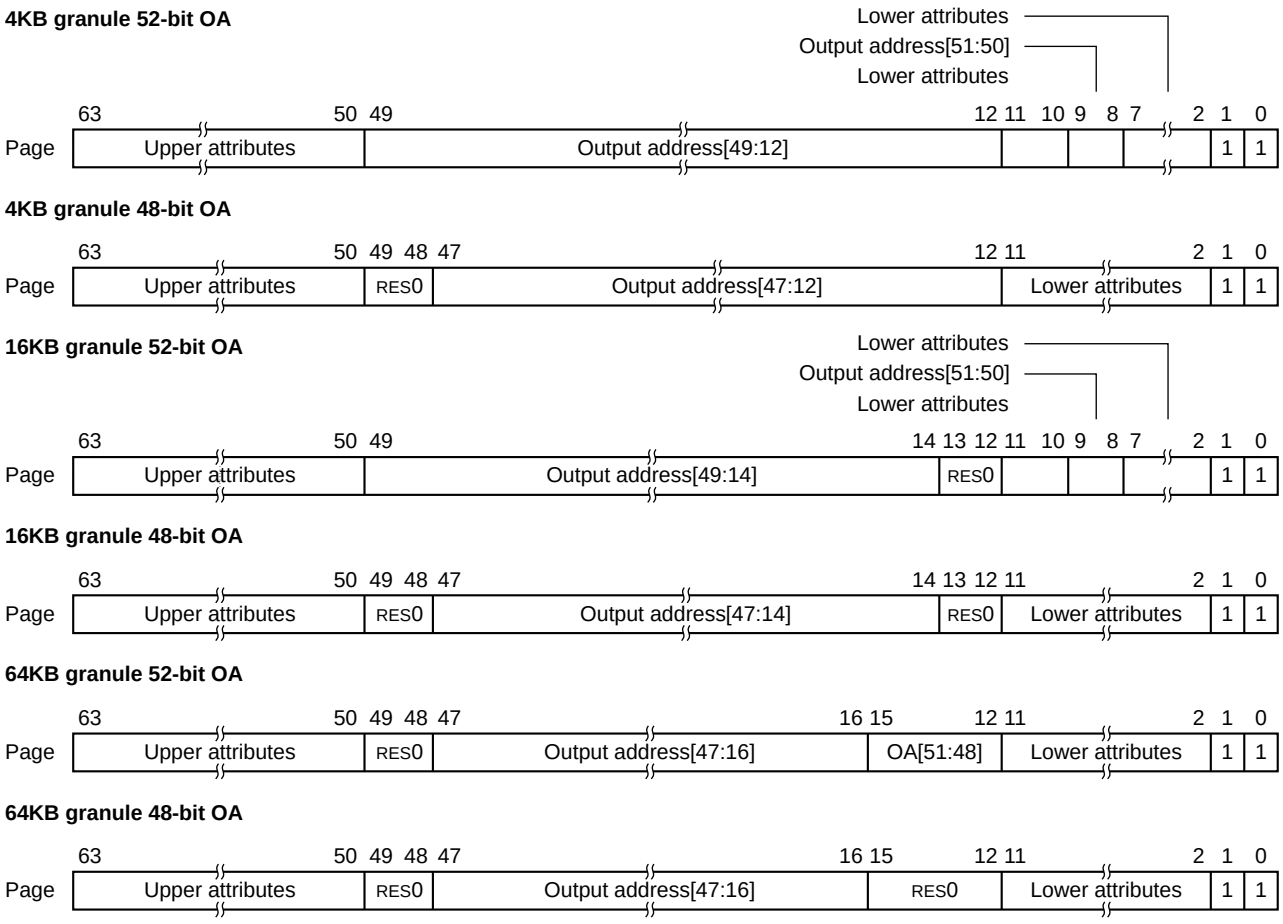


Figure D5-9 VMSAv8-64 Page descriptor formats

I_{GLMLD} For a stage 1 translation, the following figure shows the attribute fields in VMSAv8-64 Block descriptors and Page descriptors, split into upper attributes and lower attributes.

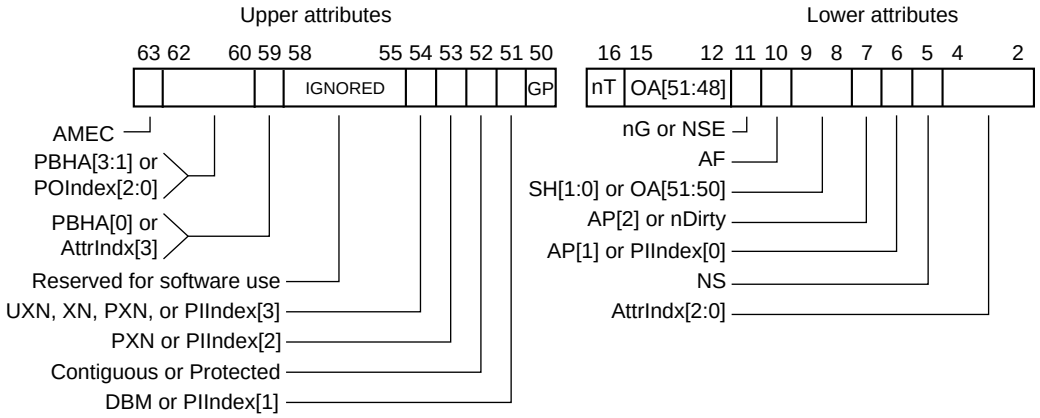


Figure D5-10 Stage 1 attribute fields in VMSAv8-64 Block and Page descriptors

R_{PBJPR} For stage 1 translations, the following table defines the fields in the VMSAv8-64 Block and Page descriptors. In this table, OAB is the OA base that is appended to the IA supplied to the translation stage to produce the final OA supplied by the translation stage.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D5-21 Stage 1 VMSAv8-64 Block and Page descriptor fields

Bit position	Field	Condition
[63]	IGNORED	Non-secure or Secure state, EL1&0 translation regime.
[62:60]	IGNORED	FEAT_HPDS2 is not implemented or PBHA is not enabled by the corresponding TCR_ELx.HWU*nn* control bit.
	PBHA[3:1]	FEAT_HPDS2 is implemented and the PBHA bit is enabled by the corresponding TCR_ELx.HWU*nn* control bit. See Page Based Hardware attributes.
	POIndex[2:0]	Stage 1 Overlay permissions enabled. See Stage 1 Overlay permissions.
[59]	IGNORED	FEAT_HPDS2 is implemented and the PBHA bit is enabled by the corresponding TCR_ELx.HWU*nn* control bit.
	PBHA[0]	FEAT_HPDS2 is implemented and the PBHA bit is enabled by the corresponding TCR_ELx.HWU*nn* control bit. See Page Based Hardware attributes.
[58:56]	Reserved for software use	-
[55]	IGNORED	-
[54]	Execute-never (XN)	The translation regime supports a single privilege level. See Stage 1 instruction execution using Direct permissions.
	Unprivileged Execute-never (UXN)	The translation regime supports two privilege levels. See Stage 1 instruction execution using Direct permissions.
	Privileged Execute-never (PXN)	The Effective value of UXN is 0. See Stage 1 instruction execution using Direct permissions
	PIIndex[3]	Stage 1 Indirect permissions enabled, regardless of other feature settings. See Stage 1 Indirect permissions.
[53]	RES0	One of the following: • The translation regime supports a single privilege level. • The EL1&0 translation regime and the Effective value of HCR_EL2.{NV, NV1} is {1, 1}. See Stage 1 instruction execution using Direct permissions
	PXN	The translation regime supports two privilege levels. See Stage 1 instruction execution using Direct permissions.
	PIIndex[2]	Stage 1 Indirect permissions enabled, regardless of other feature settings. See Stage 1 Indirect permissions.
[52]	Contiguous	The Effective value of PnCH is 0. See The Contiguous bit.
	Protected	The Effective value of PnCH is 1. See Stage 1 Protected Attribute.
[51]	Dirty bit modifier (DBM)	Stage 1 Indirect permissions are disabled. See Hardware management of the dirty state.
	PIIndex[1]	Stage 1 Indirect permissions are enabled. See Stage 1 Indirect permissions.
[50]	RES0	
[49:48]	RES0	

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bit position	Field	Condition
Block[47:17]	OAB[47:30]	The 4KB translation granule and the descriptor is a level 1 Block descriptor. Descriptor bits [29:17] are RES0.
	OAB[47:21]	The 4KB translation granule and the descriptor is a level 2 Block descriptor. Descriptor bits [20:17] are RES0.
	OAB[47:36]	The 16KB translation granule, the Effective value of TCR_ELx.DS is 1, and the descriptor is a level 1 Block descriptor. Descriptor bits [35:17] are RES0.
	OAB[47:25]	The 16KB translation granule and the descriptor is a level 2 Block descriptor. Descriptor bits [24:17] are RES0.
	OAB[47:42]	The 64KB translation granule, FEAT_LPA is implemented, and the descriptor is a level 1 Block descriptor. Descriptor bits [41:17] are RES0.
	OAB[47:29]	The 64KB translation granule and the descriptor is a level 2 Block descriptor. Descriptor bits [28:17] are RES0.
Block[16]	RES0	FEAT_BBML1 is not implemented.
	nT	FEAT_BBML1 is implemented. See Table and Block entry.
Block[15:12]	RES0	One of the following: • FEAT_LPA is not implemented. • FEAT_LPA is implemented, and the 4KB or 16KB translation granule is used.
	OAB[51:48]	FEAT_LPA is implemented and the 64KB translation granule is used.
Page[47:12]	OAB[47:12]	The 4KB translation granule is used.
	OAB[47:14]	The 16KB translation granule is used. Descriptor bits [13:12] are RES0.
	OAB[47:16]	The 64KB translation granule and FEAT_LPA is not implemented. Descriptor bits [15:12] are RES0.
	OAB[47:16]: OAB[51:48]	The 64KB translation granule and FEAT_LPA is implemented.
[11]	RES0	The translation regime supports a single privilege level and the Security state is not Root state.
	Not global (nG)	The translation regime supports two privilege levels. See Global and process-specific translation table entries.
[10]	Access flag (AF)	- See The Access flag.
[9:8]	Shareability (SH[1:0])	One of the following: • The 4KB or 16KB translation granule is used, and the Effective value of TCR_ELx.DS is 0. • The 64KB translation granule is used. See Stage 1 Shareability attributes.
	OAB[51:50]	The 4KB or 16KB translation granule is used, and the Effective value of TCR_ELx.DS is 1.
[7]	AP[2]	Stage 1 Indirect permissions are disabled. See Stage 1 data accesses using Direct permissions.
	nDirty	Stage 1 Indirect permissions are enabled. See The dirty state.
[6]	RES1	Stage 1 Indirect permissions are disabled and the translation regime supports a single privilege level.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bit position	Field	Condition
	AP[1]	Stage 1 Indirect permissions are disabled and the translation regime supports two privilege levels. See Stage 1 data accesses using Direct permissions.
	PIIndex[0]	Stage 1 Indirect permissions are enabled. See Stage 1 Indirect permissions.
[5]	RES0	The access is from Non-secure state using the EL1&0 translation regime.
	NS	The access is from Secure state.
[4:2]	AttrIndx[2:0]	See Stage 1 memory type and Cacheability attributes.
[1]	Block descriptor	0, for lookup levels less than lookup level 3.
	Page descriptor	1, for lookup level 3.
[0]	Valid descriptor	1.

D5.3.1.3 VMSAv8-64 Invalid descriptor format

RVRYZC For stage 1 translations, the following table defines the fields in the VMSAv8-64 Invalid descriptor. IGNORED fields are available for software use.

Table D5-22 Stage 1 VMSAv8-64 Invalid descriptor fields

Bit position	Field	Condition
[63:53]	IGNORED	-
[52]	IGNORED	The Effective value of PnCH is 0.
	Protected	The Effective value of PnCH is 1. See Stage 1 Protected Attribute.
[51:1]	IGNORED	-
[0]	Invalid descriptor	0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.4 Memory access control

I_{SHRMH} Fields in the descriptors, PSTATE, and system registers determine the permissions used to control access to memory and instruction execution.

R_{MNKWG} *Direct permissions* use one or more of the following to determine the base permissions that control whether or not a location can be accessed, the type of access that is permitted, and the privilege level necessary to access the location:

- The access permissions in Block descriptors and Page descriptors.
- If enabled, then the hierarchical access permissions in Table descriptors.

R_{XRLBM} *Indirect permissions* use all of the following to determine the base permissions that control whether or not a location can be accessed, the type of access that is permitted, and the privilege level necessary to access the location:

- The *Permission Indirection Index* (PIIndex) field in Block descriptors and Page descriptors.
- For each translation regime, a Permission Indirection Register (PIR) indexed by PIIndex.

D5.4.1 Stage 1 permissions

R_{TDNYB} All statements in this section apply when stage 1 uses VMSAv8-64.

R_{WBLJK} The following table lists the stage 1 permissions and the access type that is permitted when that permission is present. If the stage 1 permission is not present, that access type is not permitted.

Table D5-23 Stage 1 permissions

Stage 1 permission	Access type permitted
UnprivRead	Unprivileged data read.
UnprivWrite	Unprivileged data write.
PrivRead	Privileged data read.
PrivWrite	Privileged data write.
Execute	Instruction execution.

R_{KCFNV} If stage 1 translation is disabled, that is VTCR_EL2.MSA is 0, the PMSAv8-64 is used.

I_{KSRWD} If stage 1 translation is enabled, then stage 1 permissions are determined by Stage 1 Base permissions.
For more information, see Stage 1 Base permissions.

D5.4.1.1 Stage 1 Base permissions

I_{NHTTP} Stage 1 Direct permissions are used.

D5.4.1.2 Stage 1 Direct permissions

D5.4.1.2.1 Stage 1 data accesses using Direct permissions

R_{CBNKR} For a stage 1 translation that supports one Exception level, the following table shows the possible data access permissions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D5-24 Data access permissions for a stage 1 translation supporting one Exception level

AP[2]	Permissions
0	PrivRead, PrivWrite
1	PrivRead

R_{MGGZM} For a stage 1 translation that supports one Exception level, AP[1] is RES1.

R_{RNGJG} For a stage 1 translation that supports two Exception levels, the following table shows the possible data access permissions at EL0 and the corresponding higher Exception level.

Table D5-25 Data access permissions for a stage 1 translation supporting two Exception levels

AP[2:1]	Permissions
00	PrivRead, PrivWrite
01	PrivRead, PrivWrite, UnprivRead, UnprivWrite
10	PrivRead
11	PrivRead, UnprivRead

I_{BVWWD} If hardware management of dirty state is enabled at stage 1, then the AP[2] bit can be cleared by hardware in some situations.
For more information, see Hardware management of the dirty state.

D5.4.1.2.2 Hierarchical control of data access Direct permissions

I_{VZXYY} Translation table entries at a given lookup level can limit data access Direct permissions at subsequent lookup levels.

R_{YTXKB} For a stage 1 translation, when hierarchical control is enabled, the Table descriptor APTable[1:0] field limits the data access permissions of subsequent stage 1 translation lookup levels, regardless of the Direct permissions in subsequent lookup levels, as shown in the following table:

Table D5-26 Effect of APTable[1:0] on subsequent lookup levels

APTable[1:0]	Effect at subsequent lookup levels
00	No effect on data access permissions.
01	Removes UnprivRead and UnprivWrite.
10	Removes UnprivWrite and PrivWrite.
11	Removes UnprivRead, UnprivWrite, and PrivWrite.

I_{GQYCH} For a Permission fault, the level of the Block descriptor or Page descriptor is reported regardless of whether the lack of permissions was caused by configuration of the APTable or AP fields.

I_{PCLHM} For translation regimes that support one Exception level, APTable[0] is RES0.

I_{TPCPS} The APTable[1:0] settings are combined with the descriptor access permissions in subsequent lookup levels. They do not change the values entered in those descriptors, nor restrict what values can be entered.

I_{PKGJC} For the translation regime controlled by a TCR_ELx, one or more of the following can be used to disable the Table descriptor APTable[1:0] field so that it is IGNORED by the PE and the behavior is as if the value is 0:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the Effective value of TCR_ELx.HPD{0} is 1, hierarchical data access permission is disabled in the translation tables pointed to by TTBR0_ELx.
- If the Effective value of TCR_ELx.HPD1 is 1, hierarchical data access permission is disabled in the translation tables pointed to by TTBR1_ELx.

I_{YQWJF} The descriptor APTable field affects all subsequent lookup levels. When an APTable field is changed, software is required to use a break-before-make sequence, including TLB maintenance for all lookup levels for the VA range translated by the descriptor.

D5.4.1.2.3 Stage 1 instruction execution using Direct permissions

I_{DXKST} The instruction Execute-never, Unprivileged execute-never, and Privileged execute-never fields in a Block descriptor and Page descriptor are used to control instruction execution permissions under the Direct permissions scheme by a stage 1 translation.

R_{TRZVM} For a stage 1 translation that supports one Exception level, the XN field in the Block descriptor and Page descriptor has all of the following properties:

- If the Effective value of XN is 0, then PrivExecute is not removed by this bit.
- If the Effective value of XN is 1, then PrivExecute is removed.

R_{RBDFT} For a stage 1 translation that supports two Exception levels, the UXN field in the Block descriptor and Page descriptor has all of the following properties:

- If the Effective value of UXN is 0, then UnprivExecute is not removed by this bit.
- If the Effective value of UXN is 1, then UnprivExecute is removed.

R_{LWPVL} For a stage 1 translation that supports two Exception levels, the PXN field in the Block descriptor and Page descriptor has all of the following properties:

- If the Effective value of PXN is 0, then PrivExecute is not removed by this bit.
- If the Effective value of PXN is 1, then PrivExecute is removed.

R_{VWLLR} For stage 1 translations that support two Exception levels, if UnprivWrite is present, then PrivExecute is removed.

D5.4.1.2.4 Preventing execution from writable locations

I_{KDRYB} There are register control fields that can be used to force writable memory to be treated as XN, PXN, or UXN, regardless of the value of the corresponding descriptor fields.

R_{JLSNH} For unprivileged accesses in the EL1&0 translation regime, if SCTLR_EL1, System Control Register (EL1).WXN is 1, UnprivWrite is present, and UnprivExecute is present, then UnprivWXN is applied. Otherwise, UnprivWXN is removed.

R_{SYMMB} For privileged accesses in all translation regimes, if SCTLR_EL1, System Control Register (EL1).WXN is 1, PrivWrite is present, and PrivExecute is present, then PrivWXN is applied. Otherwise, PrivWXN is removed.

I_{TBVCH} For a stage 1 translation in a translation regime that supports two Exception levels, the corresponding SCTLR_EL1, System Control Register (EL1).WXN field does all of the following:

- If the value is 0, then there is no effect on access permissions.
- If the value is 1, then all of the following apply:
  - If the memory region has UnprivWrite, then UnprivExecute is removed, regardless of the value of the Block descriptor or Page descriptor UXN field.
  - If the memory region has PrivWrite, then PrivExecute is removed, regardless of the value of the Block descriptor or Page descriptor PXN field.

I_{FQVWM} For a stage 1 translation in a translation regime that applies to a single Exception level, the corresponding SCTLR_EL1, System Control Register (EL1).WXN field does all of the following:

- If the value is 0, then there is no effect on access permissions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the value is 1 and if the memory region is PrivWrite, then PrivExecute is removed, regardless of the value of the Block descriptor or Page descriptor XN field.

I_{XLDPQ} The SCTLR_EL1, System Control Register (EL1).WXN field reduces the memory footprint available for code injection attacks by introducing an invariant that writable memory regions are not executable.

I_{RNXSK} For the lifetime of a given virtual machine, Arm expects the SCTLR_EL1, System Control Register (EL1).WXN field to remain static in normal operation.

I_{PNSFD} SCTLR_EL1, System Control Register (EL1).WXN is permitted to be cached in a TLB.

D5.4.2 Effects on instruction execution permissions and restrictions on instruction fetch

I_{CSNFS} The descriptors contain fields used to control instruction execution permissions in stage 1 translations.

R_{BCSYZ} If instruction execution is not permitted by one of the following, then the memory region is execute-never:

- Stage 1 Direct permissions are in use, and XN is 1, or both UXN and PXN are 1.
- Stage 1 Indirect permissions are in use and mark the region as execute-never.
- The configuration of the EL2 MPU.

See also R_{RSXJG} and R_{SYLGW}.

R_{BFDZQ} If a memory region is execute-never at the current Exception level, then speculative instruction fetch from that memory region is prohibited.

R_{DYKWD} If a memory region is execute-never at the current Exception level, then when an attempt to execute from that memory region occurs, a Permission fault is generated at the first translation stage that determines the region is execute-never.

I_{PTMPK} If Device memory is not execute-never, then speculative instruction fetch from that memory region is permitted.

I_{ZZSZZ} For all Exception levels from which a read-sensitive memory region can be accessed, to avoid the possibility of a speculative fetch affecting that memory region, software is expected to define that memory region as execute-never.

R_{WRCYB} Except for the purposes of PSTATE.PAN, the descriptor execute-never fields do not prevent data accesses to the memory region translated by the descriptor.

D5.4.3 Effect of PSTATE on access permission

I_{LGNSH} PSTATE.PAN affects memory access permissions decoded from memory access control fields in the descriptors. PSTATE.UAO affects the privilege level of a memory access.
For more information on PSTATE, see Process state, PSTATE.

D5.4.3.1 PSTATE.PAN

I_{KTCFZ} PSTATE.PAN is used to prevent privileged data accesses to virtual memory addresses that are accessible from EL0.

R_{NCMTC} All statements in this section require implementation of FEAT_PAN.

R_{TBYDZ} For a translation regime, if one or more of the following apply, then the PSTATE.PAN bit has no effect in that regime:

- Stage 1 translation is disabled.
- Stage 1 translation supports a single privilege level.

R_{XTSQH} If the Effective value of PSTATE.PAN is 1, then a privileged data access from any of the following Exception levels to a virtual memory address that is accessible to data accesses at EL0 generates a stage 1 Permission fault:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- A privileged data access from EL1.
- If the Effective value of HCR_EL2.E2H is 1, then a privileged data access from EL2.

R_{DGQQK} If the value of PSTATE.PAN is 0, then a privileged data access is permitted from the corresponding privileged Exception level to a virtual memory address that is accessible at EL0.

I_{ZBRBM} When taking an exception from an EL in AArch64 state to an ELx, PSTATE.PAN is saved to and restored from SPSR_ELx.PAN.

I_{FNLWP} When entering and exiting Debug state, PSTATE.PAN is saved to and restored from DSPSR_EL0.PAN.

I_{GXFBR} When taking an exception to EL1, SCTLR_EL1.SPAN determines whether PSTATE.PAN is set to 1 or the value of PSTATE.PAN is unchanged.
When taking an exception to EL2, PSTATE.PAN remains unchanged.

R_{PMTWB} The PSTATE.PAN bit has no effect on all of the following:

- Instruction fetches.
- Data cache instructions, except DC ZVA, DC GZVA, and DC GVA.
- If FEAT_PAN2 is not implemented, then address translation instructions.
- If FEAT_PAN2 is implemented, then the address translation instructions other than AT S1E1RP and AT S1E1WP.

I_{ZRSWJ} Software can access PSTATE.PAN using all of the following instructions:

- Software can use MSR (immediate) `PAN, #Imm4` to modify PSTATE.PAN.
- Software can use MSR (register) `PAN, Xt` to modify PSTATE.PAN.
- Software can use MRS `Xt, PAN` to read PSTATE.PAN.

I_{GYGGV} If the PE is in Debug state, then a debugger can use the DRPS instruction to modify PSTATE.PAN.

D5.4.3.2 PSTATE.UAO

I_{WZGDY} PSTATE.UAO is used to control whether, for unprivileged memory access instructions, memory accesses described as unprivileged by the instruction are treated as privileged or unprivileged.

R_{NMGWS} All statements in this section require implementation of FEAT_UAO.

R_{ZVNJS} Unprivileged memory access instructions are all of the following:

- The instructions described in Load/store unprivileged.

R_{NDDVQ} If the value of PSTATE.UAO is 1, then a load/store unprivileged instruction executed at any of the following Exception levels is subject to the same access permissions as the corresponding load/store register instruction:

- EL1.
- If the Effective value of HCR_EL2.TGE is 1, EL2.

I_{CVRWF} When an exception occurs, all of the following apply to PSTATE.UAO:

- When taking an exception to ELx, PSTATE.UAO is saved to SPSR_ELx.UAO and then set to 0.
- When returning from an exception at ELx, PSTATE.UAO is restored from SPSR_ELx.UAO.

I_{QWJDP} When entering and exiting Debug state, PSTATE.UAO is saved to and restored from DSPSR_EL0.UAO.

I_{JQRCG} Software can access PSTATE.UAO using all of the following instructions:

- Software can use MSR (immediate) `UAO, #Imm4` to modify PSTATE.UAO.
- Software can use MSR (register) `UAO, Xt` to modify PSTATE.UAO.
- Software can use MRS `Xt, UAO` to read PSTATE.UAO.

I_{VCMLT} If the PE is in Debug state, a debugger can use the DRPS instruction to modify PSTATE.UAO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.4.4 Controlling memory access Security state

I_{RLRVY} The Non-secure, NS, bit in Block descriptors and Page descriptors, and the Non-secure table, NSTable, bit in Table descriptors are used to provide control over accesses from Secure state to Secure and Non-secure memory.

R_{PRTVS} In this section and subsections, *IPA or PA space* refers to the OA space of stage 1 translations.

R_{KZLVM} For a Secure translation regime, if the translation table entry is in Secure IPA or PA space, then the Block descriptor and Page descriptor NS determines all of the following:

- If NS is 0, an access to the OA specified by the descriptor is to Secure IPA or PA space.
- If NS is 1, an access to the OA specified by the descriptor is to Non-secure IPA or PA space.

R_{SKGDK} For all of the following, if a Block or Page descriptor has NS set to 1, then the OA is in the Non-secure PA space.

R_{FKJXR} For a Secure translation regime, when the descriptor is fetched from Non-secure IPA or PA space, all of the following apply to the descriptor:

- The descriptor NS and NSTable bits exist but they are SBZ instead of RES0.
- The OA or next-level table base address specified by the descriptor is in Non-secure IPA or PA space.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.5 Hardware updates to the translation tables

I_{FNBGK} All of the following determine whether hardware can update the descriptor access flag and dirty state:

- The features implemented and enabled by the PE.
- The descriptor type.
- The translation regime.
- The translation stage.

I_{CGGMJ} Hardware updates to the translation tables are subject to memory ordering requirements and restrictions on the memory types.

D5.5.1 The Access flag

R_{XFXTY} The AF in a Block descriptor and Page descriptor indicates one of the following:

- If the value is 0, then the memory region has not been accessed since the value of AF was last set to 0.
- If the value is 1, then the memory region has been accessed since the value of AF was last set to 0.

R_{DWZCQ} Descriptors with AF set to zero can never be cached in a TLB.
For more information about when translation table entries are permitted to be cached in a TLB, see Translation Lookaside Buffers.

D5.5.1.1 Software management of the Access flag

I_{CDQDG} If the AF is not managed by hardware, software management of the AF is required.

R_{SWTYY} For an implementation that does not manage the AF in hardware, when an attempt is made to translate an address using a descriptor with an AF of 0, an Access flag fault is generated.

I_{FDYZF} For an implementation that does not manage the AF in hardware, when an Access flag fault is generated, software is expected to set the AF to 1 in the descriptor that generated the fault.

I_{KQMYT} Setting the AF to 1 prevents the Access flag fault from being generated the next time an attempt is made to translate an address using that descriptor.

I_{HQBRF} When software sets the AF to 1, there is no requirement to perform TLB invalidation after setting the flag because entries with an AF set to 0 are never held in a TLB.

I_{BMRWH} If a system incorporates components that can autonomously update translation table entries that are shared with the PE, then any changes by system software to translation table entries with an AF of 0 should avoid the possibility of simultaneous updates. This excludes changes to the AF value. For example, this can be done by using a Load-Exclusive/Store-Exclusive loop or an atomic compare-and-swap operation.

D5.5.1.2 Hardware management of the Access flag

R_{WMGCW} All statements in this section require implementation of the OPTIONAL feature, FEAT_HAFDBS.

R_{LFTXR} When the PE performs a hardware update of the AF, it sets the AF to 1 in the corresponding descriptor in memory, in a coherent manner, using an atomic read-modify-write of that descriptor.

R_{HDHQG} For a translation stage, if the value of the corresponding TCR_ELx.HA is 1, then AF hardware management is enabled.

I_{BDRJG} For each translation stage, AF hardware management can be enabled independently.

R_{HDTRB} If AF hardware management is enabled, then hardware updates the AF when a memory access is made using a Block descriptor or Page descriptor and all of the following apply:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The access does not generate an Alignment fault based on the memory type, or a Permission fault.
- If AF hardware management was disabled or not implemented, the access would have generated an Access flag fault.

R_{QSPMS} If AF hardware management is enabled, then it is CONSTRAINED UNPREDICTABLE whether hardware updates the AF when a memory access is made using a Block descriptor or Page descriptor and all of the following apply:

- The access generates an Alignment fault based on the memory type, a Permission fault, or any fault that has a lower priority than a Permission fault from that stage of translation.
- If AF hardware management were disabled or not implemented, the access would have generated an Access flag fault.

I_{RGQLZ} If hardware does not update the AF from 0 to 1, then the descriptor is not permitted to be cached in a TLB.

R_{VNHWC} If AF hardware management is enabled in one or both translation stages, then when a memory access occurs, one or more of the following can be updated:

- The stage 1 AF.
- Each of the stage 2 AFs used to translate accesses to the stage 1 translation table walk and the OA.

I_{CVNQR} For stage 1 translations, if AF hardware management is enabled, then when stage 2 translation does not permit access to the OA returned by the stage 1 translation, it is permitted, but not required, for the stage 1 AF to be updated.

I_{MYQJN} For stage 2 translations, if AF hardware management is enabled, then when stage 1 translation does not permit access, it is permitted, but not required, for the stage 2 AF to be updated.

R_{LHQRX} If AF hardware management is enabled and a speculative access is successfully translated, then it is permitted, but not required, for hardware to update the AF.

R_{GGYFS} When a speculative access occurs, the rules determining the AF hardware update apply.

I_{GXGPN} When the translation of an architecturally executed memory access occurs, the architecture requires that AF is set to 1.

I_{LLCCB} When a translation occurs, the AF is permitted to be set to 1 even in the case where no architecturally executed memory access occurs. This could be due to a speculative memory access.

I_{PJWQY} When hardware updates of the Access Flag are enabled, it is permitted to update the Access Flag speculatively. This is not affected by the granule protection check on the output address of the translation.

R_{XQSMX} For an address translation stage, if AF hardware management is enabled, then when an address translation instruction is executed, all of the following apply:

- Hardware is permitted, but not required, to update the AF in the Block or Page descriptor that resolves the translation.
- If hardware attempts to update the AF and the update generates a fault, it is IMPLEMENTATION DEFINED whether this is reported as a Data Abort or not.
- If a Data Abort does not occur, then the instruction reports a result in the PAR as if the descriptors used to translate the address have an AF value of 1.

I_{ZWQRX} For more information, see Using break-before-make when updating translation table entries.

I_{KJBFT} If an implementation does not generate Access flag faults for cache maintenance by VA instructions, it is not required to perform hardware updates of AF for those cache maintenance instructions.
See also MMU faults generated by cache maintenance operations.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.5.2 The dirty state

I_{HTJWZ} The dirty state is used to indicate a memory block or page has been modified. When hardware update of the dirty state is enabled, the descriptor DBM field indicates whether the descriptor is a candidate for hardware updates of the dirty state.

R_{LRMSH} For the purpose of FEAT_HAFDBS dirty state management, a Block descriptor or Page descriptor can be described as having one of the following states:

- Non-writable.
- Writable-clean.
- Writable-dirty.

R_{XSTDV} If a Block descriptor or Page descriptor is not writable-clean and not writable-dirty, then it is described as *non-writable*.

R_{CLBGT} For a translation stage using Indirect permissions, software or hardware is permitted to manage the dirty state.

R_{DQQCV} For a translation stage using Direct permissions, only hardware is permitted to manage the dirty state.

R_{VRNVZ} For more information on managing the dirty state in hardware, see Hardware management of the dirty state.

I_{CPTSY} For a stage 1 translation using Indirect permissions at ELx, if the Effective value of TCR_ELx.HD is 0, then dirty state is managed by software.

R_{QXCRT} For stage 1 translations using Indirect permissions that grant write permission, the following table shows the descriptor state based on the Block descriptor and Page descriptor nDirty field:

nDirty field	Descriptor state
0	Writable-dirty
1	Writable-clean

Note

The polarity of the nDirty field matches the polarity of the AP[2] field used by Direct permissions.

R_{PPGMC} For write accesses translated by a writable-clean descriptor, if the descriptor dirty state is managed by software, then a Permission fault due to dirty state is generated. For information on the priority of a Permission fault due to dirty state, see Prioritization of Permission faults.

D5.5.2.1 Hardware management of the dirty state

R_{QRTKG} All statements in this section require implementation of the OPTIONAL feature, FEAT_HAFDBS, which supports both hardware update of the Access flag and hardware update of the dirty state.

I_{FLYZP} For stage 1 translations, if the Effective value of the corresponding TCR_ELx.HD is 1, then dirty state hardware management is enabled.

I_{DLNSK} If a translation stage uses Direct permissions and hardware update of the dirty state is enabled, then the descriptor DBM field indicates whether the descriptor is a candidate for hardware updates of the dirty state.

R_{PZFQC} For translations using Direct permissions, if the DBM field in a Block descriptor or Page descriptor is 0, then all of the following apply:

- For write accesses translated by the descriptor, the generation of Permission faults are unaffected by FEAT_HAFDBS.
- Hardware does not update the dirty state.

R_{XZFQH} For each translation stage using Direct permissions, if all of the following apply, then a Block descriptor or Page descriptor is described as *writable-clean*:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Dirty state hardware update is enabled at that translation stage.
- The descriptor DBM field is 1.
- For a descriptor used by stage 1 translation, the AP[2] field is 1 and this is the only reason that a Permission fault would be generated due to a write access if dirty state hardware management were disabled.

R_{BRFGY} For each translation stage using Direct permissions, if all of the following apply, then a Block descriptor or Page descriptor is described as *writable-dirty*:

- Dirty state hardware update is enabled at that translation stage.
- The descriptor DBM field is 1.
- For a stage 1 translation, the AP[2] field is 0.

I_{HFBCN} If a translation stage uses Indirect permissions, then dirty state hardware management is not separately determined by each Block descriptor and Page descriptor. This is different from the case where a translation stage uses Direct permissions, and the DBM field indicates whether the descriptor is a candidate for dirty state hardware updates.

I_{BFDYL} Only if a descriptor is writable-clean, then hardware can update the dirty state.

R_{TYCNM} For write accesses translated by a writable-clean descriptor, if the descriptor dirty state is managed by hardware, then a Permission fault due to dirty state is not generated.

R_{NSXRD} For write accesses translated by a writable-clean descriptor, if all of the following apply, then hardware updates that descriptor to writable-dirty:

- The descriptor dirty state is managed by hardware.
- The only reason that a Permission fault would be generated due to that write access would be if dirty state hardware management were disabled.

R_{PQJJR} For a translation stage, when the dirty state is updated by hardware, the PE updates the corresponding descriptor in memory, in a coherent manner, using an atomic read-modify-write of that descriptor to change the descriptor from writable-clean to writable-dirty.

R_{PRHKD} If the requirements to update both the Access flag and dirty state by hardware are met, the PE is permitted to update the Access flag to 1 as part of the same atomic read-modify-write of the descriptor that performs the dirty state update.

I_{DSHLJ} For a translation stage, Access flag faults are reported with higher priority than Permission faults. Therefore, it is not permitted for hardware to update the dirty state so that a descriptor is writable-dirty but has the Access flag clear.

R_{SGJBL} When the dirty state is updated by hardware, the PE is required to check that any copy of the descriptor cached in a TLB is not stale with regard to the descriptor in memory.

I_{YZSVV} If the descriptor has changed, then the PE is required to use the new information from the descriptor in memory, regardless of whether the modified descriptor has a different OA, different attributes, or generates a fault.

I_{WZRPC} For performance purposes, when the dirty state is updated by hardware, the PE should update or invalidate any TLB entry that contains a previously-cached copy of the descriptor. This is to prevent multiple writes by the PE from causing multiple updates to the descriptor.

I_{HQBGN} Arm expects many implementations to cause a translation table walk to occur when dirty state hardware management causes an update of a translation table descriptor.

R_{RNMLQ} If a write access translated by a writable-clean descriptor is performed architecturally, then hardware updates the dirty state of that descriptor.

R_{QDRWL} If an instruction that generates more than one single-copy atomic memory access has a fault on some, but not all, of those memory accesses, then all of the following apply:

- All accesses that do not fault are permitted to cause hardware updates of the dirty state.
- For faulting accesses that meet the requirements for hardware updates of the dirty state, those accesses are also permitted to cause hardware updates of the dirty state.

I_{BSLWF} For more information, see:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- MMU fault prioritization from a single address translation stage.
- Faults and watchpoints.

D5.5.2.2 Implications of enabling the dirty state management mechanism

R_{YWCDD} All statements in this section require implementation of the OPTIONAL feature, FEAT_HAFDBS, which supports both hardware update of the Access flag and hardware update of the dirty state.

R_{TTLCV} For stage 1 translations, translations using a descriptor in the writable-clean state are considered writable for the purposes of the corresponding SCTLR_ELx.WXN control.

I_{NSYYW} For stage 1 translations, if the corresponding SCTLR_ELx.WXN is 1, then all of the following apply:

- For a translation regime that supports a single privilege level, translations using a writable-clean descriptor are treated as execute-never.
- For a translation regime that supports two privilege levels, translations using a privileged writable-clean descriptor are treated as privileged execute-never.
- For a translation regime that supports two privilege levels, translations using an unprivileged writable-clean descriptor are treated as unprivileged execute-never.

R_{NKLQN} For a translation regime that supports two privilege levels, translations using an unprivileged writable-clean descriptor are treated as privileged execute-never.

R_{RKMHW} For the following instructions that require write permission, if the address specified in the instruction is translated by a writable-clean descriptor and hardware management of Dirty state is enabled for the corresponding stage of translation, then the descriptor is considered to grant write access and the hardware does not update the Dirty state of that descriptor:

- Data cache invalidation instruction, DC IVAC.
- Data cache and instruction cache maintenance instructions that are affected by FEAT_CMOW.
- Address translation instructions, AT S12E0W, AT S12E1W, AT S1E0W, AT S1E1W, AT S1E2W, AT S1E3W.

R_{MKSBQ} For all of the following cases, if the stage 1 descriptor is writable-clean, then it is IMPLEMENTATION DEFINED whether hardware updates the dirty state:

- A Store-Exclusive instruction to the memory location fails because the Exclusives monitor is not in the exclusive state.
- A synchronous External abort is generated on a write to the memory location.
- A Watchpoint exception is generated on a write to the memory location.

R_{KLVVN} A hardware update of the dirty state performed by one PE does not require invalidation of any corresponding TLB entries in other PEs.

R_{TXGHB} For a CAS or CASP instruction to an address translated by a writable-clean descriptor, if the comparison fails and the location is therefore not updated, then it is CONSTRAINED UNPREDICTABLE whether hardware updates the dirty state.

D5.5.3 Ordering of hardware updates to the translation tables

R_{MKHJS} A hardware update to the translation table that is caused by a load or a store, including an atomic instruction, is guaranteed to be observed, to the extent required by the Shareability attributes:

- Before a load or store, including an atomic instruction, to an arbitrary address, other than the address of the translation table entry, that appears in program order after the load or store, including an atomic instruction, causing the update to the translation table entry only if a DSB with the appropriate Shareability attributes, where the DSB applies to both loads and stores, is executed between the load or store, including an atomic instruction, that caused the update to the translation table and the subsequent load or store.
- Before a load to the translation table entry that is being updated that appears in program order after the load or store, including an atomic instruction, causing the update to the translation table entry only if a DSB with the appropriate Shareability attributes, where the DSB applies to both loads and stores, is executed between the load or store, including an atomic instruction, that caused the update to the translation table and the subsequent load.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Before a store or atomic access to the translation table entry that is being updated that appears in program order after the load or store, including an atomic instruction, causing the update to the translation table entry.
- Before a cache maintenance instruction to an arbitrary address appearing in program order after the load or store, including an atomic instruction, causing the update to the translation table entry only if a DSB with the appropriate Shareability attributes, where the DSB applies to both loads and stores, is executed between the load or store, including an atomic instructing that caused the update to the translation table entry and the subsequent cache maintenance instruction.

R_{PLLLM} An update to the translation table that is caused by a load is not ordered with respect to the load itself.

R_{FSZNW} An update to the translation table that is caused by a store or an atomic access is observed by all observers, to the extent required by the Shareability attributes, before the store itself in the case that the store is to the same location as the translation table update.

R_{RZYCN} An update to the translation table that is caused by a store or an atomic access is not ordered with respect to the store itself in the case that the store is not the same location as the translation table update.

D5.5.4 Restriction on memory types for hardware updates to translation tables

I_{YTWWW} A translation table can be placed in Normal memory with any Cacheability.

I_{XPRCJ} If hardware update of translation table entries is enabled, software is required to ensure that the translation table is located in memory that supports atomic read-modify-write updates in a coherent manner.

I_{MYHXH} Atomicity properties can only be met at the system level, and some system implementations might not support this functionality in all memory regions, including all of the following:

- Any system memory type that does not support hardware cache coherency.
- Non-cacheable memory, or memory that is treated as Non-cacheable, in an implementation that does not support hardware cache coherency.

I_{PCHYY} The system implementation determines which memory type is treated as Non-cacheable.

R_{WLTNL} All of the following Conventional memory types architecturally guarantee that hardware updates of the translation tables are atomic:

- Inner Shareable, Inner Write-Back, Outer Write-Back Normal memory.
- Outer Shareable, Inner Write-Back, Outer Write-Back Normal memory.

R_{NNNMB} If a translation table hardware update is not atomic as observed by other agents that can access memory, then the update can have one or more of the following effects:

- A synchronous External abort on the translation table walk is generated.
- The instruction generates an SError exception.
- An MMU fault is generated, reported as an Unsupported atomic hardware update using one of the following Fault status codes:
  - For Data Abort exceptions, ESR_ELx.DFSC = `0b110001`.
  - For Instruction Abort exceptions, ESR_ELx.IFSC = `0b110001`.
  - For an abort on a write to the Statistical Profiling buffer, PMBSR_EL1.FSC = `0b110001`.
- The hardware updates occur and all of the following apply:
  - There is no guarantee that the memory accesses are done atomically in regard to accesses to memory by other agents.
  - The instruction might also generate an SError exception.

R_{VSXXT} If a translation table hardware update is not atomic as observed by other agents that can access memory, and the update generates an MMU fault, all of the following apply:

- For the EL1&0 translation regime, one of the following:
  - If the atomic hardware update is not supported in the stage 1 translation due to the defined memory type, then this exception is a stage 1 abort and is taken to EL1.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - In all other cases, the exception is taken to EL2.
- For a translation stage, the MMU fault priority is at an IMPLEMENTATION DEFINED point between all of the following:
  - Immediately before the priority of an Access flag fault that is generated at the same translation stage.
  - Immediately after the priority of a Permission fault that is generated at the same translation stage.

I_{FGVJW} For more information, see Possible implementation restrictions on using atomic instructions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.6 Memory region attributes

I_{SGDKK} The memory region attributes in a descriptor define the memory type, Cacheability, and whether the memory region is Shareable. For translation regimes that support two translation stages, the stage 1 and stage 2 memory region attributes are combined to produce a resultant memory type and Cacheability, and Shareability.

I_{XCFWR} All of the following apply to the memory region attributes:

- The stage 1 OA attributes are derived from a combination of the Block descriptor and Page descriptor AttrIndx[3:0] field, and the MAIR_ELx and MAIR2_ELx register fields.
- The stage 1 translation table walk attributes are derived from the TCR_ELx.{IRGN*n*, ORGN*n*, SH*n*} fields.
- The SCTLR_EL1, System Control Register (EL1).{C, I} fields might override stage 1 attributes.
- For the EL1&0 translation regime the HCR_EL2.{CD, DC} fields might override stage 1 attributes.
- All of the following HCR_EL2.FWB values affect the attributes passed to the EL2 MPU:
  - When the Effective value of HCR_EL2.FWB is 0, memory attributes are derived from the MemAttr[3:2] descriptor bits, then combined with stage 1 attributes.
  - When the Effective value of HCR_EL2.FWB is 1, memory attributes are derived from the MemAttr[2:0] descriptor bits.
- For Device memory and Normal Non-cacheable mappings from stage 1, normalization of the Shareability that is input to the memory attributes is IMPLEMENTATION DEFINED.

I_{LHJDF} The effects of combining memory attributes defined by the hypervisor are expected to be functionally transparent to the Guest OS.

D5.6.1 Stage 1 memory type and Cacheability attributes

R_{HHGNL} If FEAT_AIE is not implemented, all of the following apply:

- For a stage 1 translation, the Block descriptor and Page descriptor AttrIndx[2:0] field holds the value *n* used to select the 8-bit field MAIR_ELx.Attr<*n*> that specifies the memory region attributes of the descriptor OA.
- It is IMPLEMENTATION DEFINED how AttrIndx[2:0] is used to index into AMAIR_ELx.

I_{FZMTR} For the memory region specified by a Block descriptor and Page descriptor, the MAIR_ELx.Attr<*n*> or MAIR2_ELx.Attr<*n*> field selected by the descriptor AttrIndx[2:0] field defines all of the following:

- The Device or Normal memory type.
- For Device memory, any one of the following Device memory types:
  - Device-nGnRnE.
  - Device-nGnRE.
  - Device-nGRE.
  - Device-GRE.
- For Normal memory, all of the following:
  - For both inner and outer Cacheability, one of the Non-cacheable, Write-Through, or Write-Back attributes.
  - For Write-Through Cacheable and Write-Back Cacheable regions, the Read-Allocate and Write-Allocate hints, each of which is Allocate or No Allocate, and the Transient allocation hints.

For more information, see Memory region tagging types.

I_{NSVYS} The memory attributes obtained from MAIR_ELx.Attr<*n*>, MAIR2_ELx.Attr<*n*>, AMAIR_ELx.Attr<*n*>, and AMAIR2_ELx.Attr<*n*> are permitted to be cached in a TLB.

I_{QNNGV} For some translation regimes, the memory region attributes determined from translation tables, MAIR_ELx, and MAIR2_ELx values might be overridden by the configuration of SCTLR_EL1, System Control Register (EL1).{C, I} and HCR_EL2.DC.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.6.2 Stage 1 Shareability attributes

R_{ZWYSH} For Normal Cacheable memory the SH[1:0] field in a stage 1 translation Block descriptor and Page descriptor encodes the Shareability attributes of the descriptor OA, as shown in the following table:

Table D5-28 Stage 1 Shareability attributes

SH[1:0]	Normal memory
00	Non-shareable
01	Reserved, CONSTRAINED UNPREDICTABLE as described in Reserved values in System and memory-mapped registers and translation table entries.
10	Outer Shareable
11	Inner Shareable

R_{PYFVQ} If a region is mapped as Device memory or Normal Non-cacheable memory after all enabled translation stages, then the region has an effective Shareability attribute of Outer Shareable.

D5.6.3 Stage 2 memory type and Cacheability attributes

R_{WYQVN} For any stage 2 translation, if the final memory type any is Normal Cacheable type, then all of the following apply:

- If the output of stage 1 specifies a Cacheable memory type, then the final cache allocation hints are the stage 1 cache allocation hints.
- If the output of stage 1 does not specify a Cacheable memory type, then the final cache allocation hints are Read Allocate, Write Allocate.

R_{KTSYP} Stage 2 translation configuration does not assign cache allocation hints.

I_{BBJDH} If the memory type from a stage 2 translation causes a stage 1 translation table walk to a Device memory type, then one of the following occurs:

- If HCR_EL2.PTW is 0, then the translation table walk occurs as if it is to Normal Non-cacheable memory, and the walk can be done speculatively.
- If HCR_EL2.PTW is 1, then the memory access generates a stage 2 Permission fault.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.7 Other descriptor fields

D5.7.1 Table and Block entry

R_{DFJNG} All statements in this section require implementation of FEAT_BBML1.

I_{GTPVH} The nT bit is supported in the VMSAv8-64 Block descriptors.

I_{XPRKH} Setting the nT bit in a Table descriptor or Block descriptor guarantees that, when changing the table or block size, accesses translated by the translation table entry do not break coherency, ordering guarantees or uniprocessor semantics, or fail to clear the Exclusives monitors.

R_{MRRPW} When using a Table descriptor or Block descriptor with the nT bit set to 1, all of the following apply:

- It is IMPLEMENTATION DEFINED whether a Translation fault is generated.
- If a Translation fault is not generated, a TLB conflict abort can be generated.

For more information, see Support levels for changing table or block size.

I_{DXRJK} If the nT bit in a Table descriptor or Block descriptor is 1, then the translation performance can be significantly impacted.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.8 Address tagging

I_{XWNGB} When address tagging is enabled, the top eight bits of the VA are ignored for the purposes of address translation and they are instead described as an address tag. An address tag might be used for software purposes, a Logical Address Tag, or both.

R_{LVTFP} If address tagging is enabled in a translation regime, then the top eight bits of the VA, bits[63:56], have all of the following properties:

- Bits[63:56] are the address tag.
- The bits are ignored during address translation.
- If the translation system is enabled, then the bits are ignored when determining whether the address is out of range and therefore generates a Translation fault.
- If the translation system is not enabled, then the bits are ignored when determining whether the address is out of range and therefore generates an Address size fault.

D5.8.1 Address tag control

I_{CKLSG} Address tagging is controlled using the *Top Byte Ignore* (TBI) field in the TCR_ELx register.

R_{BLBYZ} For a stage 1 translation that supports a single VA range, TCR_ELx.TBI{0} determines whether address tagging is enabled.

R_{LPCSW} For a stage 1 translation that supports two VA ranges, all of the following determine whether address tagging is enabled:

- TCR_ELx.TBI0 determines whether the lower address range uses address tags.
- TCR_ELx.TBI1 determines whether the upper address range uses address tags.

R_{KYBQL} The TCR_ELx.TBI{*n*} bit controls address tagging whether or not the corresponding stage 1 translation is enabled.

I_{PRTHR} If FEAT_PAuth is implemented and a TCR_ELx.TBI{*n*} bit enables address tagging, then the corresponding TCR_ELx.TBID{*n*} bit determines whether address tagging applies to both instruction addresses and data addresses, or just data addresses. For more information, see Pointer authentication.

D5.8.2 Effect of address tagging on the PC

I_{WKJBM} This section describes how the top byte of an address is considered when address tagging is enabled and an address is propagated to the PC for a branch, procedure return, exception, or exception return occurs.

R_{BBQKQ} For a stage 1 translation that supports a single VA range, if address tagging for instruction accesses is enabled in an Exception level, ELx, then bits[63:56] of the address loaded into the PC are forced to `0x00` in all of the following cases:

- A branch or procedure return within ELx.
- An exception is taken to ELx.
- When an exception return does not generate an Illegal exception return, any exception return to ELx.
- Exiting from Debug state to ELx.

R_{FWHLV} For a stage 1 translation that supports two VA ranges, if address tagging for instruction accesses is enabled in an Exception level, ELx, then bits[63:56] of the address loaded into the PC are a sign-extension of address bit[55] in all of the following cases:

- A branch or procedure return within ELx.
- An exception is taken to ELx.
- When an exception return does not generate an Illegal exception return, any exception return to ELx.
- Exiting from Debug state to ELx.

R_{WGTRC} When address tagging is enabled and an address causes a Data Abort or a *Watchpoint exception*, the address tag is included in the VA returned in the FAR.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.9 Pointer authentication

I_{XDNDT} AArch64 state supports signing the contents of a 64-bit general-purpose register that contains an address with a *pointer authentication code* (PAC), and authenticating the contents of a register with a PAC before the register is used as the target of an indirect branch, or the base register of a load or store instruction.

R_{STHJF} All statements in this section and subsections require implementation of FEAT_PAuth.

R_{JZTGK} Pointer authentication is supported only in AArch64 state.

I_{PKPFN} If Pointer authentication is implemented, then both of the following are true:

- Exactly one of ID_AA64ISAR1_EL1.GPI, ID_AA64ISAR1_EL1.GPA, or ID_AA64ISAR2_EL1.GPA3 fields is nonzero.
- Exactly one of ID_AA64ISAR1_EL1.API, ID_AA64ISAR1_EL1.APA, or ID_AA64ISAR2_EL1.APA3 fields is nonzero.

Otherwise, all of these ID fields are zero.

I_{GMTCR} Pointer authentication functionality is the same whether address translation is enabled or disabled, but is useful only when address translation is enabled.

D5.9.1 PAC field

I_{KQZPC} The pointer authentication mechanism treats the upper bits of a pointer in a 64-bit general-purpose register as a PAC field.

I_{BSXBR} The TCR_ELx.{T0SZ, T1SZ} fields are used to configure the size and location of the PAC field.

R_{DLVDN} When a 64-bit general-purpose register, Xd, holds an address with a PAC field, the location of the PAC field is determined by all of the following:

- The `bottom_PAC_bit` is 64-TCR_ELx.T*n*SZ, and the value of *n* is determined as follows:
  - If inserting a PAC into an address and address tagging is not used, then the value of Xd[63] is the value of *n*.
  - Otherwise, the value of Xd[55] is the value of *n*.
- If address tagging is used, then all of the following apply:
  - If `bottom_PAC_bit` is > 52 and the stage 1 translation supports a single VA range, then the PAC field is Xd[55:`bottom_PAC_bit`].
  - Otherwise, the PAC field is Xd[54:`bottom_PAC_bit`].
- If address tagging is not used and logical address tagging is used, then all of the following apply:
  - If `bottom_PAC_bit` is > 52 and the stage 1 translation supports a single VA range, then the PAC field is Xd[63:60, 55:`bottom_PAC_bit`].
  - Otherwise, the PAC field is Xd[63:60, 54:`bottom_PAC_bit`].
- If both address tagging and logical address tagging are not used, then all of the following apply:
  - If `bottom_PAC_bit` is > 52 and the stage 1 translation supports a single VA range, then the PAC field is Xd[63:`bottom_PAC_bit`].
  - Otherwise, the PAC field is Xd[63:56, 54:`bottom_PAC_bit`].

For more information, see Address tagging.

I_{VZRCW} In the case where address tagging is not used, then any use of a VA that has differing values of bit [63] and bit [55] is a non-canonical VA that will generate a fault.

I_{HLQVM} If the TCR_ELx.TBID{*n*} bit is used to restrict address tagging to data addresses, then instruction addresses can use Xd[63:56] bits to produce larger pointer authentication code fields.

I_{DJNGK} For all of the following reasons, if FEAT_PAuth2 is implemented, FEAT_FPAC is not implemented, and stage 1 translation is disabled, then Arm recommends not setting the TCR_ELx.T*n*SZ values to indicate an address range that is smaller than the PA size:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- For some PAC values, if TCR_ELx.T*n*SZ is set to indicate an address range that is smaller than the PA size, the address generated by a failed PAC authentication might be an address within the PA size because the upper address bits, those between the PA size and the size indicated by the TCR_ELx.T*n*SZ field, are taken from the result of the authentication process.
- Accessing memory using such an address that has failed PAC authentication does not generate an Address size fault.
- The memory access is done using upper address bits, those between the PA size and the size indicated by the TCR_ELx.T*n*SZ field, taken from the result of the authentication process.

R_{VMZGK} If the value in TCR_ELx.T*n*SZ is outside the permitted range, then one of the following CONSTRAINED UNPREDICTABLE results occur:

- The TCR_ELx.T*n*SZ value configures the `bottom_PAC_bit` value.
- The minimum permitted TCR_ELx.T*n*SZ value configures the `bottom_PAC_bit` value.
- The maximum permitted TCR_ELx.T*n*SZ value configures the `bottom_PAC_bit` value.

R_{FCMCH} If a PE treats an out of range TCR_ELx.T*n*SZ value as the maximum permitted field value or the minimum permitted field value for all purposes except reading the field value, then that behavior also applies to determining `bottom_PAC_bit`.

D5.9.2 PAC generation and verification keys

I_{RSXKN} For pointer authentication, all of the following 128-bit PAC keys are provided:

- Instruction address PAC key A, APIAKey_EL1, is the concatenation of the register values APIAKeyHi_EL1:APIAKeyLo_EL1.
- Instruction address PAC key B, APIBKey_EL1, is the concatenation of the register values APIBKeyHi_EL1:APIBKeyLo_EL1.
- Data address PAC key A, APDAKey_EL1, is the concatenation of the register values APDAKeyHi_EL1:APDAKeyLo_EL1.
- Data address PAC key B, APDBKey_EL1, is the concatenation of the register values APDBKeyHi_EL1:APDBKeyLo_EL1.
- Generic authentication PAC key A, APGAKey_EL1, is the concatenation of the register values APGAKeyHi_EL1:APGAKeyLo_EL1.

I_{KBYGG} For an Exception level, all of the following bits in the SCTLR_ELx registers are used to enable PAC generation and validation:

- SCTLR_EL1, System Control Register (EL1).EnIA enables instruction address pointer authentication using the `APIAKey_EL1` key.
- SCTLR_EL1, System Control Register (EL1).EnIB enables instruction address pointer authentication using the `APIBKey_EL1` key.
- SCTLR_EL1, System Control Register (EL1).EnDA enables data address pointer authentication using the `APDAKey_EL1` key.
- SCTLR_EL1, System Control Register (EL1).EnDB enables data address pointer authentication using the `APDBKey_EL1` key.

I_{GMQNP} All Exception levels use the same set of registers to hold PAC keys.

I_{GLRSR} When switching between Exception levels, software is expected to apply all of the following:

- Software is expected to switch the PAC keys between Exception levels.
- Software is expected to not leave the values of the current keys present in memory, typically done by zeroing those locations after switching.

D5.9.3 PAC instructions

I_{VXGGK} When a 64-bit general-purpose register is used as the target of an indirect branch instruction or the base register in a load instruction, all of the following instructions can be used to authenticate the register contents and prepare the register to be used:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Instructions that insert a PAC into the PAC field.
- Instructions that extract the value in the PAC field and check that the value is correct.
- Instructions that remove the PAC field in the register and replace it with the pointer sign-extension bits, without verification.

I_{KCKSC} An instruction that inserts a PAC into the PAC field of a 64-bit general-purpose register generates the PAC using the value of the 64-register and one or two other 64-bit diversifier values.

I_{WCZPC} FEAT_PAuth_LR provides instructions that allow signing of the 64-bit value in LR against the value of SP and a PC-relative instruction address.

R_{TMSKZ} An instruction that inserts a PAC into the PAC field of a 64-bit general-purpose register, operates on the PAC field in one of the following ways:

- If FEAT_PAuth2 is not implemented, then the pointer authentication mechanism replaces the PAC field with the PAC.
- If FEAT_PAuth2 is implemented, then the pointer authentication mechanism exclusive-ORs the bits in the PAC field with the PAC.

R_{TFBFT} When FEAT_PAuth2 is not implemented and when performing a PAC operation on a non-canonical address then the PAC is modified to make it incorrect, and one of the following applies:

- If FEAT_EPAC is implemented, then the pointer authentication mechanism sets the PAC field to 0.
- If FEAT_EPAC is not implemented, then the pointer authentication mechanism inverts one of the bits of the PAC as follows:
  - When address tagging is not used, bit [62] is inverted.
  - Otherwise, bit [54] is inverted.

I_{KBKGF} An instruction that extracts the PAC from the PAC field and checks that the value is correct does all of the following:

- The check is based on the value of the register and one or two other 64-bit diversifier values.
- If the value is correct, the PAC is replaced with the extension bits.
- If the value is incorrect, all of the following occur:
  - The PAC is replaced with the extension bits.
  - Two extension bits are set to a unique, fixed value, such that the 64-bit value represents a non-canonical VA. This is referred to as making the VA non-canonical.

I_{RZFZG} Multiple versions of pointer authentication instructions support different use cases. Some instructions combine a pointer authentication operation with another operation.

R_{WNKCJ} If PAC generation and validation is disabled, then the PACGA and `XPAC*` instructions are not affected and are always enabled.

R_{WDYDC} If PAC generation and validation is disabled, then all of the following apply:

- Instructions that insert a PAC into the address in a 64-bit general-purpose register execute as `NOPs`.
- Instructions that authenticate a PAC execute as `NOPs`.
- If an instruction combines pointer authentication with another operation, then all of the following apply:
  - The pointer authentication does not occur.
  - The combined operation behaves the same as the equivalent instruction that does not do pointer authentication.

I_{RHMHV} All of the following are examples of the resulting behavior of instructions that combine pointer authentication with another operation, if PAC generation and validation is disabled:

- A `RETAA` instruction operates as a RET instruction.
- A `LDRAA Xt, [Xn, #<simm10>]!` instruction operates as a `LDR Xt, [Xn, #<simm10>]!` instruction.

I_{ZVGYK} The PACGA instruction generates a 32-bit PAC from two 64-bit values and a generic key.

I_{ZPGRZ} The PACGA instruction can be used to protect small memory blocks. PACGA instructions can be chained to protect an arbitrary-sized memory block.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{PXWQC} For pointer authentication instructions other than the PACGA instruction, the PAC is generated using one of the following IMPLEMENTATION DEFINED methods:

- If FEAT_PACQARMA5 is implemented, then ID_AA64ISAR1_EL1.APA is nonzero and the QARMA5 block cipher algorithm is used.
- If FEAT_PACIMP is implemented, then ID_AA64ISAR1_EL1.API is nonzero and an IMPLEMENTATION DEFINED algorithm is used.
- If FEAT_PACQARMA3 is implemented, then ID_AA64ISAR2_EL1.APA3 is nonzero and the QARMA3 block cipher algorithm is used.

I_{WXMCQ} For the PACGA instruction, the PAC is generated using one of the following IMPLEMENTATION DEFINED methods:

- If FEAT_PACQARMA5 is implemented, then ID_AA64ISAR1_EL1.GPA is nonzero and the QARMA5 block cipher algorithm is used.
- If FEAT_PACIMP is implemented, then ID_AA64ISAR1_EL1.GPI is nonzero and an IMPLEMENTATION DEFINED algorithm is used.
- If FEAT_PACQARMA3 is implemented, then ID_AA64ISAR2_EL1.GPA3 is nonzero and the QARMA3 block cipher algorithm is used.

R_{ZYPJV} For a PAC generated using the `ComputePAC()` pseudocode function, where the PAC generation uses an IMPLEMENTATION DEFINED algorithm then the IMPLEMENTATION DEFINED algorithm uses the same arguments as the `ComputePAC()` function.

For a PAC generated using the `ComputePAC2()` pseudocode function, where the PAC generation uses an IMPLEMENTATION DEFINED algorithm then the IMPLEMENTATION DEFINED algorithm uses the same arguments as the `ComputePAC2()` function.

R_{JTRCS} For a set of arguments passed to the `ComputePAC()` or `ComputePAC2()` pseudocode function, the same result is produced by all PEs that an execution thread could migrate between.

R_{DPXKM} The `ComputePAC2()` pseudocode function generates a PAC for 64-bit data, using a 128-bit PAC key and two diversifiers.

R_{MLLDB} If `ComputePAC2()` is implemented using FEAT_PACQARMA5 or FEAT_PACQARMA3, then the two diversifiers are concatenated into a single 64-bit diversifier as follows:

- Bits[31:0] of the 64-bit diversifier hold the value modifier1<35:4>.
- Bits[63:32] of the 64-bit diversifier hold the value modifier2<36:5>.

I_{RNXPW} Arm strongly recommends that software at EL0 does not mix pointer authentication mechanisms. For example, using the PACGA instruction to create a PAC and using other pointer authentication instructions to authenticate, or using an instruction which uses a specific key to sign a pointer and using an instruction which uses a different key to authenticate that pointer.

D5.9.4 Faulting on pointer authentication

I_{XBGSY} A PAC authentication failure for a given VA can cause a fault to be generated in the following three manners, according to the type of the instruction and whether FEAT_FPAC and FEAT_FPACCOMBINE are implemented:

- The PAC instruction makes the VA non-canonical, such that a subsequent use of the VA generates a fault. In this case, the PAC instruction does not directly generate the fault.
- The PAC instruction makes the VA non-canonical and uses that VA such that a fault is generated by that instruction.
- The PAC instruction directly generates a PAC Fail exception, with EC code `0b011100`.

R_{MLWGL} If an instruction is a combined instruction that includes pointer authentication, then when the PAC is incorrect in a given VA, one of the following behaviors occurs:

- For a combined authenticate and load instruction, then:
  - If FEAT_FPACCOMBINE is not implemented, the VA is made non-canonical and then used as the address for the load.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - If FEAT_FPACCOMBINE is implemented, then the instruction generates a PAC Fail exception, with EC code `0b011100`.
- For a combined authenticate and branch instruction, then:
  - If FEAT_FPACCOMBINE is not implemented, the VA is made non-canonical and the PC is updated to this non-canonical value.
  - If FEAT_FPACCOMBINE is implemented, then the instruction generates a PAC Fail exception, with EC code `0b011100`.

R_{KJSRQ} For a PAC authentication instruction, `AUT*`, when the PAC is incorrect for a given VA, one of the following behaviors occurs:

- If FEAT_FPAC is not implemented, the VA is made non-canonical.
- If FEAT_FPAC is implemented, then the instruction generates a PAC Fail exception, with EC code `0b011100`.

R_{XXFXH} Predicting the result of a PAC Authentication is a form of control flow prediction.

R_{GKVYD} Prediction of the results of a PAC Authentication must adhere to the requirements of FEAT_FPACC_SPEC, if implemented.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.10 Memory aborts

R_{SQLMR} For a VMSAv8-64 implementation, all of the following abort mechanisms can cause the PE to take an exception due to a failed memory access:

- A *Debug exception* generated by the self-hosted debug system. For more information, see AArch64 Self-hosted Debug.
- An *Alignment fault* generated by a memory access that is not aligned as required by the operation. For more information, see Alignment support.
- An *MMU fault* generated by the fault checking sequence in the appropriate translation regime. For more information, see MMU fault types.
- An *External abort* generated by the memory system and not by self-hosted debug, alignment checking, or MMU checking. For more information, see External aborts.

R_{XXMXW} For AArch64 state, MMU faults are synchronous exceptions that are reported as one of the following:

- Data Aborts due to data memory access faults.
- Instruction Aborts due to instruction fetch faults.

For more information, see Synchronous exception types.

I_{WFYSR} In some cases, an MMU fault generated as part of `AT` instruction execution is reported in PAR_EL1 rather than as a synchronous abort.

I_{YMZWZ} When an MMU fault is generated and reported as an Instruction Abort or Data Abort, the Exception level that the exception is taken to depends on the translation regime and translation stage that caused the fault.

I_{DHWPX} When a memory access abort generates an exception, all of the following registers are used to record context information:

- A Fault Address Register, FAR_ELx.
- An Exception Syndrome Register, ESR_ELx.

I_{XLTCG} When an MMU fault is generated, the fault context saved in the ESR_ELx at the Exception level that the fault is taken to depends on all of the following:

- Whether the fault is reported as an Instruction Abort or as a Data Abort.
- Whether the exception is taken from the same or a lower Exception level.

I_{FVNYG} For an Instruction Abort, only the following two exceptions have a higher priority in the exception priority hierarchy:

- Software Step exception.
- PC alignment fault exception.

I_{SRPXB} For a Data Abort, only Watchpoint exceptions have a lower priority in the exception priority hierarchy.

D5.10.1 MMU fault types

R_{YLSQG} All of the following MMU fault types are supported, and there are no other MMU fault types:

- Alignment fault on a data access. For more information, see Alignment support.
- Translation fault.
- Address size fault.
- Synchronous External abort on a translation table walk.
- Access flag fault.
- Permission fault.
- TLB conflict abort.

R_{GKZPW} When an attempt to access Device memory generates an MMU fault, no memory access occurs.

R_{TQDJS} If the memory attributes cannot be determined and the resultant memory region could be Device memory, then when an attempt to access that region generates an MMU fault, no memory access occurs.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.10.1.1 Translation fault

R_{DYNYQ} If descriptor bit[0] is 0 in any translation table entry, the descriptor is invalid. When a translation table lookup returns an invalid descriptor, a Translation fault is generated and descriptor bits[63:1] are ignored.

R_{KRMJZ} If descriptor bit[1] is 0 at a lookup level that does not support Block descriptors, then the descriptor encoding is reserved. When a translation table lookup returns that descriptor, a Translation fault is generated.

R_{FDQJL} If FEAT_BBML1 is implemented, and the Block descriptor has the nT bit set, then the implementation is permitted to generate a Translation fault.

I_{RRSLQ} For any translation lookup level, when a Translation fault is generated, the reported fault code indicates the lookup level.

R_{VZZSZ} When one or more of the following apply, a level 0 Translation fault is generated on the relevant translation stage:

- The IA does not map onto a TTBR_ELx address range.
- If the IA maps onto the TTBR0_ELx address range and the IA contains any one bits above the configured IA size as determined by TCR_ELx.T0SZ.
- If the IA maps onto the TTBR1_ELx address range and the IA contains any zero bits above the configured IA size as determined by TCR_ELx.T1SZ.
- When a TLB miss occurs, the corresponding TCR_ELx.EPD*n* field prevents a translation table walk using TTBR*n*_ELx.
- When FEAT_E0PD is implemented, the corresponding TCR_ELx.E0PD*n* field prevents unprivileged access to an address translated by TTBR*n*_ELx.

R_{MLNTS} A translation table entry that generates a Translation fault is not cached in a TLB.

I_{XFTPJ} When a Translation fault is generated, the fault handler does not have to perform TLB maintenance to remove the faulting entry.

I_{PXJFV} For more information, see MMU faults generated by cache maintenance operations.

D5.10.1.2 Address size fault

R_{CVSBH} For a translation stage, if an address in one of the following has nonzero address bits above the configured OA size, then an Address size fault is generated:

- The TTBR_ELx used in the translation.
- A translation table entry.
- The translation OA.

R_{FTZHB} An Address size fault can be generated at any translation lookup level.

R_{WDZFC} For the 64KB translation granule, if the implementation does not support 52-bit PAs, then all of the following apply:

- It is IMPLEMENTATION DEFINED whether descriptor bits[15:12] being nonzero generates an Address size fault.
- For this case, it is deprecated to not generate an Address size fault.

R_{BFHQH} When an Address size fault is generated, the reported fault code indicates one of the following:

- If the fault was generated due to the TTBR_ELx used in the translation having nonzero address bits above the OA size, then a fault at level 0.
- If the fault was not generated due to the TTBR_ELx used in the translation, then the lookup level at which the fault occurred.

R_{GGQPR} When an Address size fault is generated, the translation table entry causing the fault is not cached in a TLB.

I_{PBYZQ} When an Address size fault is generated, the fault handler does not have to perform TLB maintenance to remove the faulting entry.

I_{WWYQH} An implementation is required to ensure that the Contiguous bit does not prevent generation of Address size faults. For more information, see Architectural guarantees when the Contiguous bit is misprogrammed.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{JTTDF} For more information, see:

- MMU faults generated by cache maintenance operations.
- Output address size configuration.

D5.10.1.3 External abort on a translation table walk

R_{YLGBV} If an External abort on a translation table walk is generated, then it is reported in one of the following ways:

- If the External abort is synchronous, then it is reported as one of the following:
  - If the translation table walk is due to an instruction fetch, then a synchronous Instruction Abort exception.
  - If the translation table walk is due to a data access, an address translation instruction, or a cache maintenance instruction operating by VA, then a synchronous Data Abort exception.
- If the External abort is asynchronous, then it is reported using the SError exception.

I_{JVVQN} A TLB or intermediate TLB caching structure might support:

- An arbitrary number of translation table lookup levels.
- One or more translation stages that might not correspond to an address translation lookup stage.

R_{BGPQR} If a synchronous External abort is generated due to a TLB or intermediate TLB caching structure, including parity or ECC errors, then all of the following are permitted:

- If the PE cannot precisely determine the translation stage at which the error occurred, then it is reported and prioritized as a stage 1 fault.
- If the PE cannot precisely determine the lookup level at which the error occurred, then the lookup level is reported and prioritized as one of the following:
  - The lowest-numbered lookup level that could have caused the error.
  - If the PE cannot determine any information about the lookup level, then level 0.

D5.10.1.4 Access flag fault

I_{BMSDD} For an implementation that does not manage the AF in hardware, when and only when a non-speculative access is made to an address and an attempt is made to translate that address using a descriptor with an AF of 0, an Access flag fault is generated.

R_{LFGXS} For a translation lookup of a Block descriptor or Page descriptor, when an Access flag fault is generated, the reported fault code indicates the lookup level.

I_{PNQBP} When an Access flag fault is generated, the translation table entry causing the fault is not cached in a TLB.

I_{TQXPT} When an Access flag fault is generated, the fault handler does not have to perform TLB maintenance to remove the faulting entry.

I_{RGFRF} For more information, see:

- MMU faults generated by cache maintenance operations.
- Software management of the Access flag.

D5.10.1.5 Permission fault

R_{CSRQR} When a Permission fault is generated, the reported fault code indicates the lookup level of the Block or Page descriptor used in the translation, even if the lack of permission was caused by hierarchical permission controls.

R_{JVXRH} A translation table entry that generates a Permission fault is permitted to be cached in a TLB.

I_{WJLXV} If software updates a stage 1 translation table due to a Permission fault, then the software is required to invalidate the appropriate TLB entry to prevent stale information in a TLB from being used by a subsequent memory access.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{BXJNT} For more information, see:

- Prioritization of Permission faults.
- MMU faults generated by cache maintenance operations.
- Memory access control.
- The Protected System Memory Architecture.

D5.10.1.6 TLB conflict abort

I_{CNNYQ} When a TLB has not been properly invalidated, such as when architecturally required TLB invalidation is not done, an address lookup might hit multiple TLB entries.

R_{ZQNWZ} When a lookup address hits multiple TLB entries, it is IMPLEMENTATION DEFINED whether a TLB conflict abort is generated.

R_{WJMZJ} When a TLB conflict abort is generated, it is reported as one of the following:

- On an instruction fetch it is reported as an Instruction Abort.
- On a data access or a cache management instruction, it is reported as a Data Abort.
- For an `AT S1E0*` or `AT S1E1*` instruction executed from EL1, when a TLB conflict abort is generated on a stage 2 translation, it is reported as a Data Abort.

I_{DHRWD} For an `AT` instruction other than an `AT S1E0*` or `AT S1E1*` instruction executed from EL1, a TLB conflict does not generate an abort.

R_{QWTKL} For the EL1&0 regime, when a TLB conflict abort is generated and both stages of translation are enabled, then one of the following applies:

- If FEAT_BBML1 is implemented and the TLB conflict abort is generated due to changing the table or block size or Contiguous bit, then the abort is reported to EL2.
- Otherwise, it is IMPLEMENTATION DEFINED whether it is reported to EL1 or EL2.

R_{ZMVZT} A TLB Conflict abort cannot be generated for a disabled stage of translation.

I_{JYDXJ} When a TLB conflict abort is generated, all of the following are reported:

- A `0b110000` Fault status code.
- The lookup address that caused the fault.

D5.10.2 MMU faults generated by address translation instructions

R_{XGWVV} An address translation instruction uses the translation system and can generate all of the following faults:

- Translation fault.
- Address size fault.
- Synchronous External abort on a translation table walk.
- Access flag fault.
- Permission fault.
- TLB conflict abort.

D5.10.2.1 Exceptions to reporting the fault in PAR_EL1

R_{QBSVM} For an address translation instruction executed at a particular Exception level, ELx, if a synchronous External abort is generated on a stage 1 translation table walk, then the Data Abort exception is taken to the Exception level to which a synchronous External abort on a stage 1 translation table walk for a memory access from ELx would be taken.

R_{FDJKZ} When an address translation instruction causes a synchronous External abort that is reported as an exception, all of the following apply:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- PAR_EL1 is UNKNOWN.
- The ESR_ELx of the target Exception level indicates that the fault was generated by a translation table walk due to a cache maintenance operation or address translation instruction.
- The FAR_ELx of the target Exception level holds the VA supplied to the address translation instruction.

D5.10.3 MMU faults generated by cache maintenance operations

I_{QZTRG} There are no alignment requirements in cache maintenance instructions and execution of these instructions cannot generate an Alignment fault.

I_{BDCCY} Cache maintenance instructions treated as a NOP cannot generate MMU faults.

R_{TRTWX} It is IMPLEMENTATION DEFINED whether execution of any cache maintenance instruction by VA that is not treated as a NOP can generate an Access flag fault.

I_{ZNSBH} When an implementation can generate MMU faults on execution of cache maintenance operations, Permission faults are generated according to the type of cache maintenance operation and system register configuration.

R_{JYWZL} If a DC Invalidate by address instruction, for example, DC IVAC, DC IGVAC, or DC IGDVAC is not treated as a NOP and does not have write permission to the location it invalidates, then a Permission fault is generated.

I_{XQZJB} DC ZVA is not a cache maintenance instruction.

I_{LJLSP} For more information, see:

- Data and unified caches.
- Instruction caches.
- Memory aborts.
- Permission fault.
- A64 Cache maintenance instructions.

D5.10.4 MMU fault-checking sequence

I_{NTFDN} For a translation stage, when an IA is translated to an OA, the fault checking sequence is done using all of the following steps:

1. If the IA is subject to an alignment check, then check the alignment and do one of the following:
   - If the IA is not aligned, then an Alignment fault is generated.
   - If the IA is aligned, then continue to the next step.
2. Check that the IA maps to a translation table base address register, TTBR, and do one of the following:
   - If the IA does not map to a TTBR, then a Translation fault is generated.
   - If the IA maps to a TTBR and translation using that TTBR is disabled, then a Translation fault is generated.
   - If the IA maps to a TTBR, then get the translation table base address and continue to the next step.
3. Check that the translation table base address size is valid and do one of the following:
   - If the translation table base address size is not valid, then an Address size fault is generated.
   - If the translation table base address size is valid, then continue to the next step.
4. Fetch the descriptor, using all of the following steps:
   - The descriptor address is a PA from a stage 1 translation, then all of the following are done:
     - The fault checking sequence is done by the EL2 MPU, using the PA as an IA.
     - The OA from the MPU access is the PA.
5. Check that the descriptor is valid and do one of the following:
   - If the descriptor is not valid, then a Translation fault is generated.
   - If the descriptor is valid, then continue to the next step.
6. Check that the descriptor address size is valid and do one of the following:
   - If the descriptor address size is not valid, then an Address size fault is generated.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the descriptor address size is valid, then continue to the next step.

7. Check the descriptor type and do one of the following:
   - For a Table descriptor, get the descriptor address, get the hierarchical permissions and attributes that apply to subsequent lookup levels, and go back to step 4 to fetch the next descriptor.
   - For a Block descriptor or Page descriptor, continue to the next step.
8. If FEAT_BBML1 is implemented, and the fetched descriptor is a block descriptor with the nT bit set, then the implementation can generate a Translation fault.
9. If AF hardware management is disabled or not implemented, check the descriptor AF bit and do one of the following:
   - If AF is 0, then an Access flag fault is generated.
   - If AF is 1, then continue to the next step.
10. If AF hardware management is enabled, then the hardware attempts to update the AF and that might result in a Permission fault from the EL2 MPU, or a Synchronous External abort.
11. Get the OA and OA space from the Block descriptor or Page descriptor returned by the translation table walk.
12. Check the alignment required for the output memory type and do one of the following:
    - If the OA alignment is not valid, then an Alignment fault is generated.
    - If the OA alignment is valid, then continue to the next step.
13. Check the OA space access permissions and do one of the following:
    - If the OA space access is not permitted, then a Permission fault is generated.
    - If the OA space access is permitted, then continue to the next step.
14. If dirty state hardware management is enabled, then the hardware attempts to update the dirty state and that might result in a Permission fault at stage 2, or a Synchronous External abort.
15. The translation stage returns the OA and region attributes.

I_{MDJYB} For a translation stage, the following figure illustrates the MMU fault checking sequence.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

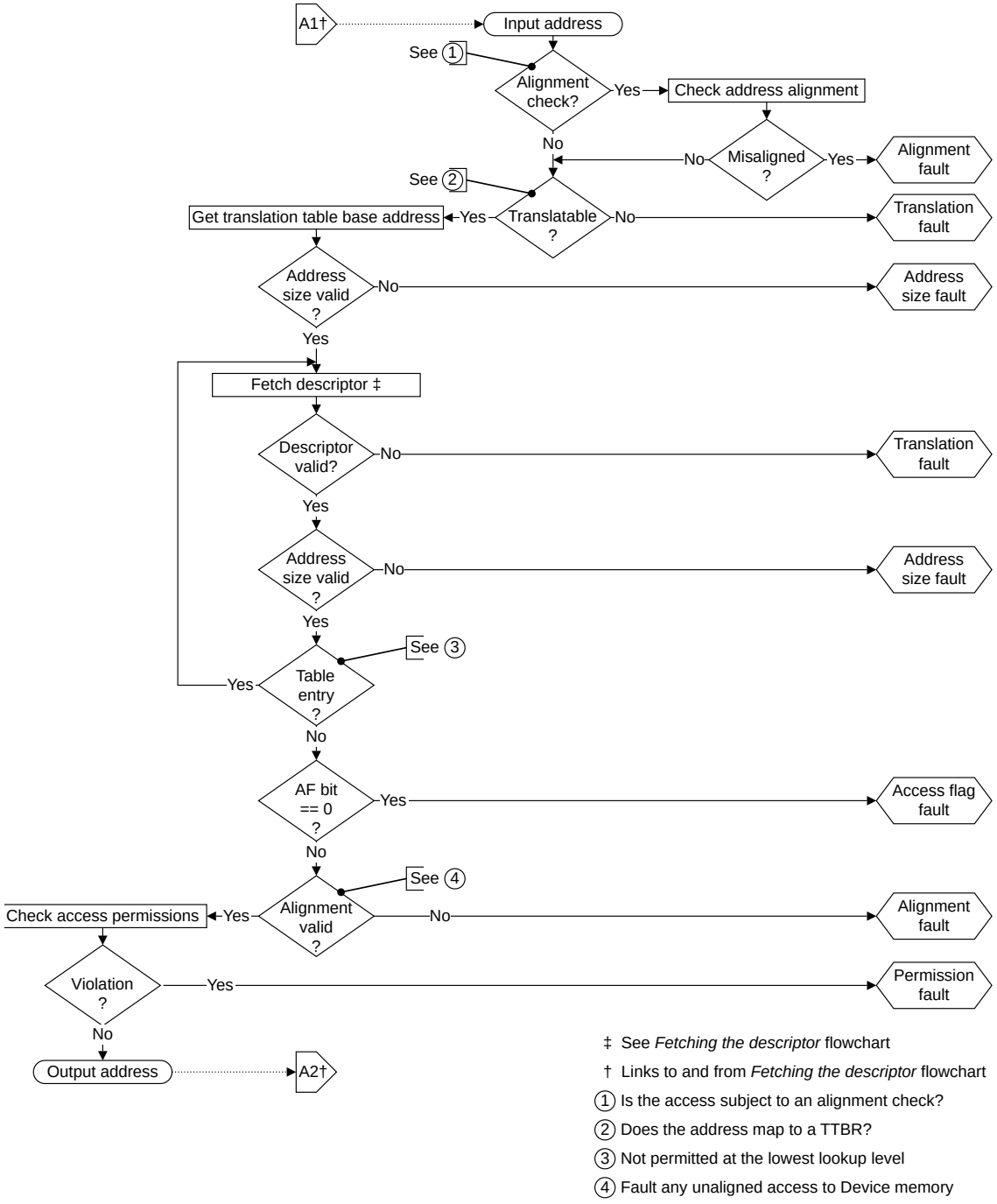


Figure D5-11 MMU fault checking sequence

I_{GFKXY} For a translation stage, the following figure illustrates the steps taken to fetch a descriptor during a translation table walk.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

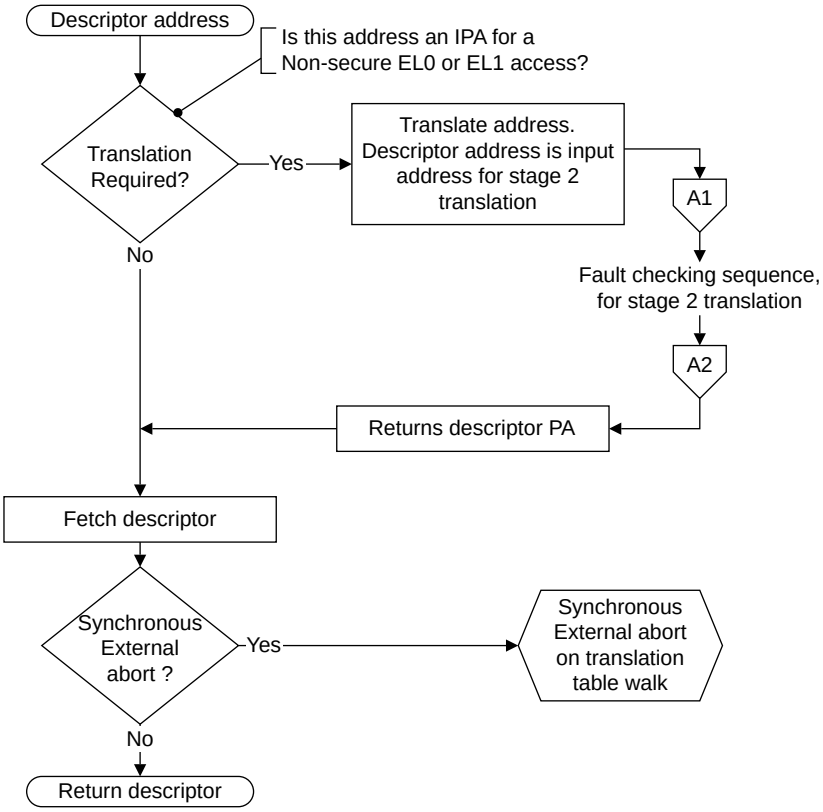


Figure D5-12 Fetching a descriptor during a translation table walk

I_{GLRRS} For more information, see Permission fault.

D5.10.4.1 The lookup level associated with MMU faults

I_{XCCCM} For MMU faults, the *Data Fault Status Code* (DFSC) and *Instruction Fault Status Code* (IFSC) in the ESR_ELx report the translation lookup level associated with a given MMU fault type.

I_{CRKPZ} When an MMU fault is generated, all of the following determine the lookup level associated with the fault:

- For a fault generated by a translation table walk, the lookup level at which the fault occurred.
- For a Translation fault, one of the following:
  - The lookup level at which the fault occurred.
  - If a fault is generated because translation table walks are disabled by TCR_ELx.EPD*n*, the IA is outside the range specified by the associated TTBR_ELx, or FEAT_E0PD is enabled and prevents access to addresses translated by TTBRn_ELx, then the fault is reported as a level 0 fault.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- For an Access flag fault, the lookup level at which the fault occurred.
- For a Permission fault, including a Permission fault generated by hierarchical permissions, the final lookup level that returned the Block descriptor or Page descriptor.

I_{CJZVP} For more information, see Permission fault.

D5.10.5 MMU fault prioritization from a single address translation stage

R_{JNVJY} For a single translation stage in a VMSAv8-64 translation regime, the following list shows the possible MMU faults on a memory access, from highest priority to lowest priority, and includes where stage 2 faults that are generated within a stage 1 translation table walk are prioritized:

1. For a stage 1 translation, an Alignment fault not caused by memory type.
2. If one or more of the following is true, then a Translation fault:
   - The IA out of the translated address range.
   - For a stage 1 translation, the IA requiring a TTBR_ELx that is disabled.
3. If one or more of the following is true, an Address size fault on the address in TTBR_ELx:
   - The address size check on TCR_EL1.IPS, TCR_EL2.{I}PS or VTCR_EL2.PS fails.
   - The programmed address is outside the implemented range.
4. If stage 2 address translation is enabled, then a stage 2 MMU fault on a level -1 memory access during a stage 1 table walk, including an Address size fault caused by a PA outside the implemented range.
5. A synchronous parity or ECC error on a level -1 lookup of a translation table walk.
6. A synchronous External abort on a level -1 lookup of a translation table walk.
7. Translation fault on a level -1 translation table entry.
8. If one or more of the following is true, an Address size fault on the address in a level -1 lookup translation table entry:
   - The address size check on TCR_EL1.IPS, TCR_EL2.{I}PS or VTCR_EL2.PS fails.
   - The OA is out of the implemented range.
9. A synchronous parity or ECC error on a level 0 lookup of a translation table walk.
10. A synchronous External abort on a level 0 lookup of a translation table walk.
11. Translation fault on a level 0 translation table entry.
12. If one or more of the following is true, an Address size fault on the address in a level 0 lookup translation table entry:
    - The address size check on TCR_EL1.IPS, TCR_EL2.{I}PS or VTCR_EL2.PS fails.
    - The OA is out of the implemented range.
13. A synchronous parity or ECC error on a level 1 lookup of a translation table walk.
14. A synchronous External abort on a level 1 lookup of a translation table walk.
15. Translation fault on a level 1 translation table entry.
16. If one or more of the following is true, an Address size fault on the address in a level 1 lookup translation table entry:
    - The address size check on TCR_EL1.IPS, TCR_EL2.{I}PS or VTCR_EL2.PS fails.
    - The OA is out of the implemented range.
17. A synchronous parity or ECC error on a level 2 lookup of a translation table walk.
18. A synchronous External abort on a level 2 lookup of a translation table walk.
19. Translation fault on a level 2 translation table entry.
20. If one or more of the following is true, an Address size fault on the address in a level 2 lookup translation table entry:
    - The address size check on TCR_EL1.IPS, TCR_EL2.{I}PS or VTCR_EL2.PS fails.
    - The OA is out of the implemented range.
21. A synchronous parity or ECC error on a level 3 lookup of a translation table walk.
22. A synchronous External abort on a level 3 lookup of a translation table walk.
23. Translation fault on a level 3 translation table entry.
24. If one or more of the following is true, an Address size fault on the address in a level 3 lookup translation table entry:
    - The address size check on TCR_EL1.IPS, TCR_EL2.{I}PS or VTCR_EL2.PS fails.
    - The OA is out of the implemented range.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

25. Access flag fault.
26. Alignment fault caused by the memory type.
    For an Alignment fault generated as a result of the conditions specified in Crossing a page boundary with different memory types or Shareability attributes or Crossing a peripheral boundary with a Device access, the prioritization of the Alignment fault is IMPLEMENTATION DEFINED, at a point between here and above Synchronous External abort on the memory access.
27. Permission fault. For more information, see Prioritization of Permission faults.
28. Synchronous parity or ECC error on the memory access.
29. Synchronous External abort on the memory access.

R_{WLJCM} For a TLB conflict abort, all of the following apply:

- The priority of the abort is IMPLEMENTATION DEFINED.
- The priority of the abort is required to be higher than any abort that depends on a value held in a TLB.

I_{HGZXP} The cause of a TLB conflict abort depends on the TLB implementation.

R_{CSFKV} For an IMPLEMENTATION DEFINED MMU fault caused by an unsupported combination of memory access type and memory type and reported with ESR_ELx.DFSC value `0b110101`, the priority is IMPLEMENTATION DEFINED.

R_{KYXHT} For an MMU fault at a translation stage caused by an unsupported atomic hardware update, the priority of this fault lies at an IMPLEMENTATION DEFINED point between all of the following:

- A priority immediately higher than an Access flag fault generated by the same translation stage as this MMU fault.
- A priority immediately lower than a Permission fault generated by the same translation stage as this MMU fault.

D5.10.5.1 Prioritization of Permission faults

R_{NRVMH} The following list shows the priority order of stage 1 Permission faults, from highest priority to lowest priority:

1. Fault due to stage 1 Base permissions.
2. Fault due to write of an Allocation Tag to a Canonically Tagged memory location.
3. Fault due to the dirty state.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.11 Translation Lookaside Buffers

I_{ZVNKM} The Arm architecture does not specify any structure of *Translation Lookaside Buffers* (TLBs), and permits any structure that complies with the requirements described in this section.

R_{XCLRD} Translation table entries that generate a Translation fault, an Address size fault, or an Access flag fault are never cached in a TLB.

R_{SQBCS} When address translation is enabled, a translation table entry for an in-context translation regime that does not cause a Translation fault, an Address size fault, or an Access flag fault is permitted to be cached in a TLB or intermediate TLB caching structure as the result of an explicit or speculative access.

R_{ZVMQM} A translation table entry is not permitted to be cached in a TLB or intermediate caching structure if a synchronous external abort on the translation table walk prevents the PE from determining if the translation table entry would otherwise cause a Translation fault, an Address size fault, or an Access flag fault.

I_{RCGCH} When address translation is enabled, if a translation table entry meets all of the following requirements, then that translation table entry is permitted to be cached in a TLB or intermediate TLB caching structure at any time:

- The translation table entry itself does not generate a Translation fault, an Address size fault, or an Access flag fault.
- The translation table entry is not from a translation regime configured by an Exception level that is lower than the current Exception level.

I_{LBGNR} The Arm architecture permits TLBs to cache certain information from System control registers, including when any or all translation stages are disabled. The individual register descriptions specify System control register fields are permitted to be cached in a TLB.
For more information, see AArch64 System Register Descriptions.

R_{JBNTF} For any System register field that is described as being permitted to be cached in a TLB, if that field takes an Effective value as a result of a System register control field at a higher Exception level, then the Effective value is also permitted to be cached in a TLB.

R_{WVTXJ} When executing at EL2 the TLB entries associated with the EL1&0 translation regime are out-of-context.

I_{RLTGW} The VMSA provides TLB maintenance instructions for the management of TLB contents.

R_{NDCMB} When a translation stage is disabled and then re-enabled, TLB entries are not corrupted.

D5.11.1 TLB behavior at reset

I_{GWZWM} When a reset occurs, an implementation is not required to automatically invalidate a TLB.

R_{BQCDZ} When a reset occurs, a TLB is affected in all of the following ways:

- All TLBs reset to an IMPLEMENTATION DEFINED state that might be UNKNOWN.
- It is IMPLEMENTATION DEFINED whether a specific TLB invalidation routine is required to invalidate a TLB before translation is enabled after a reset.

R_{HPBWC} For the ELx reset is taken to, when a reset occurs, SCTLR_EL1, System Control Register (EL1).M is reset to 0. For the translation regime controlled by that SCTLR_EL1, System Control Register (EL1).M bit, when SCTLR_EL1, System Control Register (EL1).M is 0 and SCTLR_EL2.BR is 1, the background region for the EL2 MPU is used.

R_{QNFHZ} If an implementation requires a specific TLB invalidation routine, then all of the following apply:

- The routine is IMPLEMENTATION DEFINED.
- The implementation documentation is required to clearly document the routine.
- Arm recommends that the routine is based on the TLB maintenance instructions.

I_{WVKXW} On a Cold reset or Warm reset, an implementation might require TLBs to maintain their contents from before the reset, including one or more of the following reasons:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Power management.
- Debug requirements.

I_{WZHDM} For more information on the TLB maintenance instructions used in a TLB invalidation routine, see TLB maintenance instructions.

D5.11.2 TLB lockdown

R_{SSQZC} TLB lockdown support is IMPLEMENTATION DEFINED.

R_{XXDPS} If an implementation supports TLB lockdown, then all of the following apply:

- The lockdown mechanism is IMPLEMENTATION DEFINED.
- The implementation documentation is required to clearly document the interaction of the TLB lockdown mechanism with the architecture.
- A locked TLB entry is guaranteed to remain in the TLB, unless the locked TLB entry is affected by a TLB maintenance operation.
- An unlocked TLB entry is not guaranteed to remain in the TLB.
- If a translation table entry is modified, then it is not guaranteed that a locked TLB entry remains coherent with the modified translation table entry.

I_{DXXYV} If a translation table entry is modified, then it is not guaranteed that a locked TLB entry remains incoherent with the modified translation table entry because the lockdown mechanism might permit a TLB maintenance instruction to trigger an update of the locked TLB entry.
For more information, see The interaction of TLB lockdown with TLB maintenance instructions.

R_{DSQQW} The implementation is permitted to use the reserved IMPLEMENTATION DEFINED register encodings to implement TLB lockdown functions.

I_{LSKYV} TLB lockdown functions might include, but are not limited to, all of the following:

- Unlock all locked TLB entries.
- Prefetch a translation table entry into a specific TLB level.

R_{DSLVY} If an implementation supports TLB lockdown and EL2 is enabled, then when executing at EL1 or EL0, an exception due to TLB lockdown can be routed to one of the following:

- EL1, as a Data Abort exception.
- EL2, as a Hyp Trap exception.

D5.11.3 Use of ASIDs to reduce TLB maintenance requirements

I_{LHWHR} TLB maintenance can be reduced during context switches by associating the translation table lookups from some translation regimes with an *address space identifier* (ASID).

R_{JHDZC} The rules in this section apply only to translation regimes that support two privilege levels.

R_{QRRHL} The rules in this section refer to the *controlling ELx*, in an R-Profile implementation the controlling ELx is EL1 for stage 1 translations in the E1&0 translation regime.

D5.11.3.1 Global and process-specific translation table entries

I_{FRDGD} For translation regimes that use an ASID, it is possible to mark translations as global or non-global.

R_{XXNPZ} A TLB entry associated with a specific ASID value, or a specific ASID and a VMID value, can only be used to translate a VA in a context that is associated with the same ASID value.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{WTVTJ} For stage 1 of a translation regime that supports two privilege levels, non-global TLB entries are associated with a specific ASID.

I_{LCNWN} For stage 1 of a translation regime that can support two privilege levels, Arm expects that software configures translations specific to a process to be associated with a specific ASID.

I_{HGKKC} The ASID permits software to switch between process-specific translation table mappings without removing previous mappings cached for another ASID from a TLB.

I_{LQMXG} For stage 1 of a translation regime that can support two privilege levels, each TTBR0_ELx and TTBR1_ELx has an ASID field.

R_{SCBSR} All the following values of TCR_ELx.A1 determine the ASID value that is used by the translations of both the lower VA range and upper VA range:

- If the value is 0, then the TTBR0_ELx.ASID is used.
- If the value is 1, then the TTBR1_ELx.ASID is used.

R_{PYWKY} For stage 1 of a translation regime that supports two privilege levels, TCR2_ELx.A2 determines all of the following:

- If the value is 0, then the one ASID selected by TCR_ELx.A1 is used.
- If the value is 1, then two ASIDs are used.

R_{XPSGZ} For a controlling ELx, if two concurrent ASIDs are used, then TCR_ELx.A1 does not select which ASID field is used for the purpose of caching translations in a TLB, and instead all of the following apply:

- Any non-global translation of an address in the TTBR0_ELx VA range is associated with TTBR0_ELx.ASID.
- Any non-global translation of an address in the TTBR1_ELx VA range is associated with TTBR1_ELx.ASID.

I_{NWLMM} The scope of `TLBI` operations is not affected by the Effective value of TCR2_ELx.A2.
This means that a `TLBI ASIDE1*` operation removes TLB entries associated with the specified ASID, regardless of which half of the VA range those entries would be used to translate, and would not be required to remove any other TLB entries.

I_{QYCVK} For execution at EL0, if FEAT_CSV2 is implemented, and the EL1&0 translation regime is used, the architecture defines a hardware-defined context as including an ASID. However, for execution at EL1 or at EL2 when using this translation regime, the hardware-defined context does not include an ASID.
For more information see Restrictions on the effects of speculation.

R_{SCLCL} For a controlling ELx, if two concurrent ASIDs are used, then for the purposes of FEAT_CSV2, the ASID used in the hardware-defined context when executing at EL0 is one of the following:

- If TCR_ELx.A1 is 0, then TTBR0_ELx.ASID.
- If TCR_ELx.A1 is 1, then TTBR1_ELx.ASID.

R_{QGKGF} The nG bit in a Block descriptor and Page descriptor indicates one of the following:

- If the value is 0, the translation is global and the TLB entry applies to all ASID values.
- If the value is 1, the translation is non-global and the TLB entry applies to only the current ASID value.

R_{CHVGJ} For a translation regime that supports global and non-global translations, translation table entries from lookup levels other than the final lookup level are treated as non-global, regardless of the value of the nG bit in the final lookup level.

R_{JYHZR} For a translation in Secure state, if the NSTable bit in a Table descriptor is 1 at any level of the translation table walk, then the resulting translation is treated as non-global, regardless of the nG bit value in the Block descriptor or Page descriptor for the translation.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.11.3.2 ASID size

R_{KHXQX} The ASID size is an IMPLEMENTATION DEFINED choice of 8 bits or 16 bits.

I_{NHTBK} The maximum supported ASID size is indicated by ID_AA64MMFR0_EL1.ASIDBits.

I_{BZXYN} For an implementation that supports 16-bit ASIDs, all of the following values of TCR_ELx.AS specify whether TTBRn_ELx.ASID[15:8] are used:

- If TCR_ELx.AS is 0, then TTBRn_ELx.ASID[15:8] are not used and are treated in all of the following ways:
  - For every purpose other than direct reads of TTBR0_ELx.ASID and TTBR1_ELx.ASID, the bits are ignored by hardware.
  - When used for allocating and matching entries in a TLB, the bits are treated as if they are all zeros.
- If TCR_ELx.AS is 1, then TTBRn_ELx.ASID[15:8] are used for allocating and matching entries in a TLB.

I_{ZRWYC} Bits[15:8] of the ASID field in `TLBI{P}` instructions are considered by hardware regardless of the value of TCR_ELx.AS.

D5.11.3.3 Common not private translations

I_{XJYLH} FEAT_TTCNP allows multiple PEs in the same Inner Shareable domain and operating in the same translation regime to use the same translation table entries in a given translation stage.

R_{FWQJZ} If all of the following conditions are met, translation table entries pointed to by TTBR_ELx are shared with all other PEs in the Inner Shareable domain, and are referred to as common not private translation tables:

- For the PEs that share the translation tables, the Effective value of the corresponding TTBR_ELx.CnP is 1.
- For the PEs that share the translation tables, the same translation regime in the same Security state applies to the corresponding TTBR_ELx.
- If an ASID applies to the corresponding translation stage, then all of the PEs that share non-global translation table entries are required to have the same current ASID.

R_{LGDJT} For a common not private translation table, if a System register field with all of the following characteristics is set by a particular PE to a value that is different than the value set by the other PEs, then it is CONSTRAINED UNPREDICTABLE whether the System register field is interpreted using the value of the particular PE or the value of one of the other PEs that are sharing the translation table entry:

- The register field applies to the translation stage.
- The register field is permitted to be cached in a TLB.

I_{YVLVD} The TTBR_ELx.CnP bit is permitted to be cached in a TLB.

R_{QLGWZ} For a common not private translation table, if a TTBR_ELx does not point to the same translation table as the other TTBR_ELx registers, then the system is misconfigured and an address translation using that TTBR_ELx causes one of the following to occur:

- Multiple hits in a TLB, which is permitted to generate a TLB conflict abort.
- A CONSTRAINED UNPREDICTABLE result due to caching of control or data values.

For more information, see TLB conflict abort and CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.12 TLB maintenance

R_{RVJDB} For the purpose of TLB maintenance, a TLB entry is any structure that holds a translation table entry, including intermediate TLB caching structures and temporary working registers in translation table walk hardware.

I_{NDFND} When a translation table walk occurs, translation table entries that are permitted to be cached in a TLB might be held in TLB caching structures.

R_{PDGZY} Entries held in a TLB are distinguished by all of the following context information:

- The Security state.
- The translation regime.
- If applicable to the translation regime, then whether the translation is global or non-global.
- If the translation is non-global, then the ASID.

R_{FFWJK} TLBs, or TLB caching structures, are not guaranteed to remain coherent with changes to translation table entries, and are therefore distinct from data caches.

R_{ZYZYK} Indexing an intermediate TLB structure by the IA is permitted.

R_{BKKRB} If FEAT_nTLBPA is not implemented, then indexing an intermediate TLB structure by all of the following is also permitted:

- The PA of the location holding the translation table entry.
- For stage 1 translations, the IPA of the location holding the translation table entry.

R_{TJQVP} If FEAT_nTLBPA is not implemented, any TLB maintenance instruction that applies to a PE with the context information that is relevant to the translation table entry ensures cached copies of translation table entries are invalidated for that PE in all of the following implementations of intermediate TLB structures:

- Indexing the TLB using the PA of the location holding the translation table entry.
- For stage 1 translations, indexing the TLB using the IPA of the location holding the translation table entry.

I_{PFNFJ} FEAT_nTLBPA permits software to determine the existence of intermediate TLB caching structures that are indexed by PA or IPA and perform TLB maintenance accordingly when it would otherwise not be required.

R_{LGSCG} When a TLB maintenance instruction targets only stage 1 entries, all of the following apply:

- The maintenance applies to any entries in caching structures that include stage 1 information used to translate the address or range of addresses being invalidated.
- If the TLB maintenance targets a specific ASID, then entries in caching structures that are not tagged with the specific ASID are not invalidated.
- If the stage 1 translation information contained in a single Block or Page has been collectively cached as one or more smaller TLB entries, for example as the result of splintering, then all entries containing that stage 1 information are invalidated, regardless of whether the entry would be used to translate the address being invalidated.

R_{KVQYF} It is permissble to cache stage 1 VMSAv8-64 and stage 2 PMSAv8-64 attributes as a common entry for the Secure EL1&0 translation regime.

R_{GYQRW} Visibility of stage 2 PMSAv8-64 MPU updates for stage 1 VMSAv8-64 contexts requires the associated TLB invalidation for stage 2. The stage 2 invalidation is not required to apply caching structures that combine stage 1 and stage 2 attributes.

R_{GPPYH} When a translation table entry associated with a specific ASID is modified, software is required to invalidate the corresponding TLB entry to ensure that the modified translation table entry is visible to subsequent execution, including speculative execution.

R_{VNRFW} When a System register field is modified and that field is permitted to be cached in a TLB, software is required to invalidate all TLB entries that might be affected by the field, at any address translation stage in the translation regime even if the translation stage is disabled, using the appropriate ASID, after any required System register synchronization. For more information, see Synchronization requirements for AArch64 System registers.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{WZCBG} When a translation table entry that generates a Translation fault, Address size fault, or Access flag fault is changed to one that does not fault, all of the following apply to software:

- TLB invalidation is not required because an entry that generates one of the listed faults is never cached in a TLB.
- A Context synchronization event is required to ensure that the completed change to the translation table entry affects subsequent instruction fetches.

D5.12.1 Using break-before-make when updating translation table entries

R_{VSPZR} If multiple execution threads use the same translation tables, then when a translation table entry is modified in one or more of the following ways, the architecture requires software to use a break-before-make sequence:

- A change to the memory type, Shareability or Cacheability.
- The translation OA is changed and one or more of the following apply:
  - Either or both of the old and new translations grant write permission, including cases where the DBM bit is set and hardware updates of the dirty state is enabled.
  - The memory contents at the new OA do not match the memory contents at the previous OA.
- A global entry is created that might overlap with non-global entries in a TLB.

I_{WZRHR} A change to the translation OA space is considered a change to the translation OA.

I_{THWDH} Use of a break-before-make sequence to ensure that old and new translation table entries are never simultaneously visible to different execution threads is guaranteed to prevent all of the following problems:

- Creating multiple TLB entries that apply to the same address.
- The effects of TLB caching breaking coherency.
- The effects of TLB caching breaking single-copy atomicity properties.
- The effects of TLB caching breaking ordering guarantees or uniprocessor semantics.
- The effects of TLB caching causing a failure to clear the Exclusives monitors.

R_{DDMVT} A break-before-make sequence requires all of the following steps:

1. Replace the old translation table entry with an invalid entry.
2. Execute a DSB instruction to ensure the invalid entry is visible.
3. Invalidate TLB entries based on the translation table entry with a broadcast TLB invalidation instruction.
4. Execute a DSB instruction to ensure the invalidation completes.
5. Write the new translation table entry
6. Execute a DSB instruction to ensure the new entry is visible.

I_{GXGZY} For a translation stage with AF hardware management enabled, if a translation table entry is modified and the break-before-make sequence is not followed, then all of the following failures associated with AF hardware updates can occur:

- When a memory location associated with that translation table entry is accessed, the AF is not set.
- When hardware updates to that translation table entry are followed by stores appearing later in program order, the ordering required is not followed.

R_{RTBRL} For a translation stage with dirty state hardware management enabled, if a translation table entry is modified and the break-before-make sequence is not followed, then all of the following failures associated with dirty state hardware updates can occur:

- When a memory location associated with that translation table entry is not written, hardware modifies the AP[2] descriptor bit.
- When a memory location associated with that translation table entry is written, hardware does not modify the AP[2] descriptor bit.
- When hardware updates to that translation table entry are followed by stores appearing later in program order, the ordering required is not followed.

R_{LGRYM} If translation table entries are changed without appropriate TLB maintenance operations, including in the case where use of the break-before-make sequence is required but software does not follow the break-before-make sequence, it is possible that TLBs concurrently hold multiple different copies of those translation table entries.
In this situation, the following behaviors are permitted for a speculative or architectural access to the address resolved by those TLB entries:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Use of the address matches multiple entries in a TLB, and a TLB conflict abort is detected. In this case, no access is made to memory based on those TLB entries. If the access is architectural, then the TLB conflict abort is reported as an exception.
- The resulting behavior is CONSTRAINED UNPREDICTABLE, and gives a behavior consistent with translation using one of the matching entries, or an amalgamation of more than one of the matching entries, but cannot permit access to memory regions with permissions or attributes that would not be possible in the current Security state at the current Exception level. This includes, for example, insufficient TLB maintenance by Non-secure state must not permit it to access any memory in Secure PA space.

I_{HLHBH} For more information, see CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values.

D5.12.2 Support levels for changing table or block size

R_{PVTFW} All statements in this section apply only to changing table or block sizes without changing any other property requiring break-before-make as specified in the rule R_{WHZWS} in Using break-before-make when updating translation table entries.

R_{KFLJB} When a translation table entry is modified to change the table or block size, the hardware provides one of the following possible implementations affecting the break-before-make requirement to avoid breaking coherency, ordering guarantees or uniprocessor semantics, or failing to clear the Exclusives monitors:

- If FEAT_BBML1 is not implemented, then software is required to use the break-before-make sequence.
- If FEAT_BBML1 is implemented, then software can use all of the following:
  - The break-before-make sequence.
  - The nT bit in the Table descriptor or Block descriptor.
- If FEAT_BBML2 is implemented, then software can change table or block size without breaking coherency, ordering guarantees or uniprocessor semantics, or failing to clear the Exclusives monitors.

For more information, see Block translation entry.

R_{HHCXC} When using a Table descriptor or Block descriptor with the nT bit set to 1, any accesses translated do not break coherency, ordering guarantees or uniprocessor semantics, or fail to clear the Exclusives monitors.

I_{HYQMB} If the TLB entries are not invalidated after the writes that modified the translation table entries are completed, then a TLB conflict abort can be generated because in a TLB there might be multiple translation table entries that all translate the same IA.

For Table descriptors, this also applies to intermediate TLB caching structures.

For more information, see TLB conflict abort.

I_{GYJBJ} When an MMU fault is reported while changing table or block sizes but before completing the required TLBI to invalidate the old translation table entry, the syndrome information reported for the fault is consistent with one of the following:

- The old translation table structure.
- The new translation table structure.

R_{KHRBC} If FEAT_BBML1 is implemented, then changing the Contiguous bit in a set of Block descriptors or Page descriptors can be done without breaking coherency, ordering guarantees or uniprocessor semantics, or failing to clear the Exclusives monitors.

R_{FCPSG} If FEAT_BBML1 is implemented and the Contiguous bit in a set of Block descriptors or Page descriptors is changed, then a TLB conflict abort can be generated because multiple translation table entries might exist within a TLB that translates the same IA.

I_{CFFVK} If FEAT_BBML1 is implemented and a TLB conflict abort is generated, then TLB maintenance is required to remove the multiple TLB entries that translate the same address.

For Table descriptors, this also applies to intermediate TLB caching structures.

For more information, see TLB maintenance due to TLB conflict.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.12.3 TLB maintenance due to TLB conflict

R_{BLRKC} If multiple TLB entries translate the same address, then the `ALL` or `VMALL` form of a TLB maintenance instruction that targets the given translation regime is guaranteed to remove all TLB entries within that regime.

I_{CNGYR} The following instructions are guaranteed to remove the TLB entries associated with a conflict:

- `VMALLE1`.
- `ALLE1`.

R_{GRVDR} If multiple TLB entries translate the same address, then the minimum set of TLB maintenance operations required to guarantee all TLB entries associated with that address and translation regime have been invalidated is IMPLEMENTATION DEFINED.

D5.12.4 The interaction of TLB lockdown with TLB maintenance instructions

R_{VGPNS} If a TLB entry is locked and a TLB invalidate all instruction is executed and, if that entry was not locked the TLB invalidate all instruction would invalidate that entry, then one of the following IMPLEMENTATION DEFINED behaviors occur:

- The locked TLB entry is not affected.
- If the TLB entry is locked or might be locked, an IMPLEMENTATION DEFINED Data Abort exception is generated.

I_{FRJSL} The IMPLEMENTATION DEFINED Data Abort on a TLB invalidate all instruction permits TLB invalidation routines to choose to invalidate a large range of addresses, without considering whether any TLB entries are locked, or to not affect locked TLB entries.

R_{BMHZW} If a TLB entry is locked and a TLB invalidate by VA or invalidate by ASID instruction is executed and, if that entry was not locked the TLB invalidate by VA or invalidate by ASID instruction would invalidate that entry, then one of the following IMPLEMENTATION DEFINED behaviors occur:

- The locked TLB entry is invalidated.
- The locked TLB entry is not affected.
- If the TLB entry is locked or might be locked, an IMPLEMENTATION DEFINED Data Abort exception is generated.

I_{FXBBY} The exception syndrome definitions include a fault code for cache and TLB lockdown faults.

R_{BCLPC} If an implementation uses an abort mechanism when entries that can be locked down are not actually locked down, then all of the following are required:

- The IMPLEMENTATION DEFINED instruction sequences that perform the required operations on entries that are not locked down are documented.
- An IMPLEMENTATION DEFINED mechanism is documented that ensures no TLB entries are locked.
- For locked entries, one of the other specified alternatives is implemented.

I_{DRZHY} If an implementation supports TLB lockdown, then Arm recommends that IMPLEMENTATION DEFINED TLB maintenance instruction sequences use the architecturally-defined operations to minimize the number of customized operations required.

I_{PNDQN} For more information on TLB lockdown, see TLB lockdown.

D5.12.5 TLB maintenance instructions

I_{HMJDP} For a given translation regime, the architecture defines TLB maintenance instructions that provide all of the following functions:

- Invalidate all corresponding entries in a TLB.
- Invalidate a corresponding single TLB entry by ASID for a non-global entry.
- Invalidate all corresponding TLB entries that match a specific ASID.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Invalidate all corresponding TLB entries that match a specific VA, regardless of the ASID.
- Invalidate all corresponding TLB entries within a range of addresses.

I_{XMWNR} Each TLB maintenance instruction can be applied to one of the following:

- Only the PE that executes the instruction
- All PEs in the same Shareability domain as the PE that executes the instruction.

I_{JFCRF} The A64 assembly language syntax of a TLB maintenance instruction is `TLBI{P} <operation>{, <Xt>}`.

I_{KTDKY} All of the following apply to the TLB maintenance instructions:

- The `TLBI` operations take zero or one register arguments, for cases where 64 bits or fewer of scope information is required.
- The `TLBIP` operations take a pair of register arguments, for cases where more than 64 bits of scope information is required.

I_{NDJSJ} The `<operation>` in `TLBI{P} <operation>{, <Xt>}` is one of the following:

- For `TLBI` instructions:
  - `ALLE1{NXS}`, `ALLE2{NXS}`, `ALLE3{NXS}`, `ALLE1IS{NXS}`, `ALLE2IS{NXS}`, `ALLE3IS{NXS}`, `ALLE1OS{NXS}`, `ALLE2OS{NXS}`, or `ALLE3OS{NXS}`.
  - `VMALLE1{NXS}`, `VMALLE1IS{NXS}`, `VMALLE1OS{NXS}`, `VMALLS12E1{NXS}`, `VMALLS12E1IS{NXS}`, `VMALLS12E1OS{NXS}`, `VMALLWS2E1{NXS}`, `VMALLWS2E1IS{NXS}`, or `VMALLWS2E1OS{NXS}`.
  - `ASIDE1{NXS}`, `ASIDE1IS{NXS}`, or `ASIDE1OS{NXS}`.
- For `TLBI` and `TLBIP` instructions:
  - `{R}VA{L}E1{NXS}`, `{R}VA{L}E2{NXS}`, `{R}VA{L}E3{NXS}`, `{R}VA{L}E1IS{NXS}`, `{R}VA{L}E2IS{NXS}`, `{R}VA{L}E3IS{NXS}`, `{R}VA{L}E1OS{NXS}`, `{R}VA{L}E2OS{NXS}`, or `{R}VA{L}E3OS{NXS}`.
  - `{R}VAA{L}E1{NXS}`, `{R}VAA{L}E1IS{NXS}`, or `{R}VAA{L}E1OS{NXS}`.
  - `{R}IPAS2{L}E1{NXS}`, `{R}IPAS2{L}E1IS{NXS}`, or `{R}IPAS2{L}E1OS{NXS}`.

I_{YBNHL} The `<operation>` in `TLBI{P} <operation>{, <Xt>}` has a structure of `{R}<type><regime><shareability>{NXS}` with all of the following components:

- `R` is optional and determines one of the following:
  - When present, the instruction applies to all TLB entries translating addresses within the address range.
  - When not present, the instruction applies to all TLB entries translating a single address that could be used by the PE that executes the TLB maintenance instruction.
- `<type>` is one of the following:
  - `ALL` specifies all translations at `<regime>`.
  - `VMALL` specifies all stage 1 translations for the affected translation regime, and if applicable, matching the current VMID.
  - `VMALLS12` specifies all stage 1 and stage 2 translations at EL1 with the current VMID.
  - `VMALLWS2` specifies all writable-dirty stage 2 translations at EL1 with the current VMID.
  - `ASID` specifies all non-global translations for the affected translation regime with the supplied ASID.
  - `VA{L}` specifies all translations at `<regime>` using the supplied address and, if an ASID is supplied, the supplied ASID or global entries.
  - `VAA{L}` specifies all global entries and non-global entries, regardless of the ASID value, for the affected translation regime using the supplied address.
  - `IPAS2{L}` specifies all stage 2 translations using the supplied IPA.
- For the `VA{L}`, `VAA{L}`, and `IPAS2{L}` types, `L` is an optional parameter specifying that the invalidation applies only to cached entries containing translation information returned by the final lookup level of the translation table walk.
- `<regime>` is one of the following Exception levels:
  - If `E1`, then EL1.
  - If `E2`, then EL2.
- `<shareability>` is one of the following:
  - `IS` indicates that the instruction applies to all TLBs in the Inner Shareable domain.
  - `OS` indicates that the instruction applies to all TLBs in the Outer Shareable domain.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— The parameter is optional, and no value indicates that the instruction applies to all TLBs that could be used by the PE that executes the instruction.

I_{YKBMJ} If `<regime>` is higher than the current Exception level, then the TLB maintenance instruction is UNDEFINED.

I_{DJQYS} All TLB maintenance instructions are UNDEFINED at EL0.

I_{LQDRK} In an R-profile implementation, all of the following TLB instructions are UNDEFINED:

- `TLBI ALLE3`.
- `TLBI ALLE3{I}{O}S`.
- `TLBI VAE3{I}{O}{S}`.
- `TLBI VALE3{I}{O}{S}`.

I_{BWGVK} When a TLB entry is invalidated by one PE, it is inconsistent with the architecture to allow another PE to refill that TLB entry so that the new entry gives the appearance to software that the invalidation did not occur.

I_{BNSBV} The `<Xt>` in `TLBI{P} <operation>{, <Xt>}` has all of the following properties:

- It specifies a register that passes one or both of an address and an ASID as an argument.
- For operations that include a VA, if FEAT_TTL is implemented, then the register passes the level hint.
- For range-based TLB maintenance operations, then the register passes information about the address range.
- It is required by the `TLBI ASID`, `TLBI{P} {R}VA{L}`, `TLBI{P} {R}VAA{L}`, and `TLBI{P} {R}IPAS2{L}` instructions.

D5.12.5.1 TLB maintenance instructions that do not apply to a range of addresses

I_{ZVNJG} For TLB maintenance instructions that take a VA, an ASID, or both as an argument, and that do not apply to a range of addresses, the register specified by the `Xt` argument has the following format:

- Register bits[63:48] are one of the following:
  - If the instruction requires an ASID argument, the ASID.
  - If the instruction does not require an ASID argument, RES0.
- If FEAT_TTL is not implemented, then register bits[47:44] are RES0.
- If FEAT_TTL is implemented, then register bits[47:44] are one of the following:
  - If the instruction requires a VA argument, the translation table level hint, TTL.
  - If the instruction does not require a VA argument, RES0.
- Register bits[43:0] are one of the following:
  - If the instruction requires a VA argument, VA[55:12].
  - If the instruction does not require a VA argument, RES0.

R_{JGGKN} For TLB maintenance instructions that take a VA, hardware interprets VA[63:56] as each having the same value as VA[55].

I_{LHKCP} If the implementation supports 16-bit ASIDs, then software is required to set the upper 8 bits of the ASID to 0 when the context being invalidated uses only 8 bits.

I_{KTQZV} For TLB maintenance instructions that take a register argument that holds an IPA and that do not apply to a range of addresses, the register specified by the `Xt` argument has the following format:

- Register bit[63] is the NS bit specifying the Secure or Non-secure IPA space.
- If FEAT_TTL is not implemented, then register bits[62:40] are RES0.
- If FEAT_TTL is implemented, then all of the following apply:
  - Register bits[62:48] are RES0.
  - Register bits[47:44] are the translation table level hint, TTL.
  - Register bits[43:40] are RES0.
- Register bits[39:36] are one of the following:
  - If 52-bit addresses are supported, IPA[51:48].
  - If 52-bit addresses are not supported, RES0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Register bits[35:0] are IPA[47:12].

I_{HQTLK} For TLB maintenance instructions that take a register argument that holds a VA or an IPA, register bits that hold the address have all of the following properties:

- If the 4KB granule size is used, all address bits in the register bits are used by the invalidation instruction.
- If the 16KB granule size is used, register bits[1:0] are ignored because address bits[13:12] have no effect on the invalidation instruction.
- If the 64KB granule size is used, register bits[3:0] are ignored because address bits[15:12] have no effect on the invalidation instruction.

D5.12.5.2 TLB maintenance instructions that apply to a range of addresses

I_{QPHNP} For TLB maintenance instructions that take a VA, or a VA and an ASID, and that apply to a range of addresses, the register specified by the `Xt` argument has the following format:

- Register bits[63:48] are one of the following:
  - If the instruction requires an ASID argument, the ASID.
  - If the instruction does not require an ASID argument, RES0.
- Register bits[47:46] are the translation granule size, TG.
- Register bits[45:44] are the SCALE field that is the exponent element of the calculation that produces the upper range.
- Register bits[43:39] are the NUM field that is the base element of the calculation that produces the upper range.
- If FEAT_TTL is not implemented, then register bits[38:37] are RES0.
- If FEAT_TTL is implemented, then register bits[38:37] are the translation table level hint, TTL.
- Register bits[36:0] give the starting address of the range of addresses, BaseADDR.

I_{DKWRD} For TLB maintenance instructions that take a register argument that holds an IPA and that apply to a range of addresses, the register specified by the `Xt` argument has the following format:

- Register bit[63] is the NS bit specifying the Secure or Non-secure IPA space.
- Register bits[62:48] are RES0.
- Register bits[47:46] are the translation granule size, TG.
- Register bits[45:44] are the SCALE field that is the exponent element of the calculation that produces the upper range.
- Register bits[43:39] are the NUM field that is the base element of the calculation that produces the upper range.
- If FEAT_TTL is not implemented, then register bits[38:37] are RES0.
- If FEAT_TTL is implemented, then register bits[38:37] are the translation table level hint, TTL.
- Register bits[36:0] specify the starting address of the range of addresses, BaseADDR.

R_{PVVXP} For TLB instructions that apply to a range of addresses, the following table shows the TG field encodings that define the translation granule size for the translations that are being invalidated, and the Translation_Granule_Size in bytes used in determining the address range.

Table D5-29 TG field encodings in TLB instructions that apply to a range of addresses

TG	Translation granule size	Translation_Granule_Size
00	Reserved	N/A
01	4KB translation granule	4096
10	16KB translation granule	16384
11	64KB translation granule	65536

R_{XKMMX} For TLB instructions that apply to a range of addresses, if the translations use a different translation granule size than the one specified by the TG field, then the architecture does not require that the instruction invalidate those entries.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{QKRKL} For all TLB range maintenance instructions, TLB entries that translate one or more addresses within the address range determined by the following formula are invalidated:

`[BaseADDR <= input_address < BaseADDR + ((NUM +1)*2^(5*SCALE +1) * Translation_Granule_Size)].`

R_{QNPXY} For a translation regime that supports two VA ranges, if a TLB range maintenance instruction is issued with an address in the TTBR1_ELx half of the VA space, and the SCALE and NUM values cause the range to exceed the top of the address space, then the address is not considered to wrap on overflow and the PE is not required to invalidate any entries inserted for the corresponding TTBR0_ELx half of the VA space.

I_{HKYNQ} For a TLBI range maintenance instruction, `BaseADDR` is derived from bits[36:0] in the register specified by the `Xt` argument as one of the following:

- For the 4KB granule size, one of the following:
  - If the Effective value of TCR_EL1.DS is 0, then `BaseADDR[48:12]`.
  - If the Effective value of TCR_EL1.DS is 1, then `BaseADDR[52:16]`. In this case, `BaseADDR[15:12]` is treated as all zero.
- For the 16KB granule size, one of the following:
  - If the Effective value of TCR_EL1.DS is 0, then `BaseADDR[50:14]`.
  - If the Effective value of TCR_EL1.DS is 1, then `BaseADDR[52:16]`. In this case, `BaseADDR[15:14]` is treated as all zero.
- For the 64KB granule size, `BaseADDR[52:16]`.

I_{XGGRD} For all of the following, the invalidated address range is UNPREDICTABLE:

- For the 4KB translation granule, one of the following:
  - The TTL field is `0b01` and `BaseADDR[29:12]` are not all zero.
  - The TTL field is `0b10` and `BaseADDR[20:12]` are not all zero.
- For the 16KB translation granule, one of the following:
  - The TTL field is `0b10` and BaseADDR[24:14] are not all zero.
- For the 64KB translation granule, one of the following:
  - The TTL field is `0b01` and `BaseADDR[41:16]` are not all zero.
  - The TTL field is `0b10` and `BaseADDR[28:16]` are not all zero.

I_{ZNXDY} For more information, see Broadcast TLB maintenance.

D5.12.5.3 Translation table level hint

R_{BLDQZ} All statements in this section require implementation of FEAT_TTL.

I_{LTZBK} For the address being invalidated by a TLB maintenance instruction, the TTL field in the register specified by `Xt` indicates the lookup level of the translation table walk holding the leaf entry.

I_{WVDLN} Hardware can use the TTL field to determine if the page might have been splintered into multiple TLB entries.

R_{SQXYZ} For an entry being invalidated by a TLB maintenance instruction, if an incorrect TTL value is specified, then the architecture does not require any entries to be invalidated from a TLB.

I_{VWPQL} For TLB instructions that apply to a single address, the following table shows the 4-bit TTL field encodings.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D5-30 TTL field encodings in TLB instructions that apply to a single address

TTL[3:2]	Translation granule	TTL[1:0]	Information supplied
00	-	RES0	No information supplied about the translation level. Hardware assumes that the leaf entry can be from any level.
01	4KB	00	If FEAT_LPA2 is not implemented, then this value is reserved, and hardware treats this as if TTL[3:2] is `0b00`. If FEAT_LPA2 is implemented, then the leaf entry for the address being invalidated is on level 0 of the translation table walk.
		01	The leaf entry for the address being invalidated is on level 1 of the translation table walk.
		10	The leaf entry for the address being invalidated is on level 2 of the translation table walk.
		11	The leaf entry for the address being invalidated is on level 3 of the translation table walk.
10	16KB	00	This value is reserved, and hardware treats this as if TTL[3:2] is `0b00`.
		01	If FEAT_LPA2 is not implemented, then this value is reserved, and hardware treats this as if TTL[3:2] is `0b00`. If FEAT_LPA2 is implemented, then the leaf entry for the address being invalidated is on level 1 of the translation table walk.
		10	The leaf entry for the address being invalidated is on level 2 of the translation table walk.
		11	The leaf entry for the address being invalidated is on level 3 of the translation table walk.
11	64KB	00	This value is reserved, and hardware treats this as if TTL[3:2] is `0b00`.
		01	The leaf entry for the address being invalidated is on level 1 of the translation table walk.
		10	The leaf entry for the address being invalidated is on level 2 of the translation table walk.
		11	The leaf entry for the address being invalidated is on level 3 of the translation table walk.

I_VNDBJ For TLB instructions that apply to a range of addresses, the following table shows the 2-bit TTL field encodings.

Table D5-31 TTL field encodings in TLB instructions that apply to a range of addresses

TTL	Information supplied
00	The leaf entries in the range can be using any level for the translation table entries.
01	When the 4KB or 64KB translation granule is used, all leaf entries to invalidate are level 1 translation table entries. If FEAT_LPA2 is not implemented, then when using a 16KB translation granule, this value is reserved, and hardware treats this as if TTL is `0b00`. If FEAT_LPA2 is implemented, then when using a 16KB translation granule, all leaf entries to invalidate are level 1 translation table entries.
10	All leaf entries to invalidate are level 2 translation table entries.
11	All leaf entries to invalidate are level 3 translation table entries.

D5.12.5.4 TLB maintenance instruction scope

R_YPJSC If `<type>` is `ALL` in a TLB maintenance instruction, then all of the following apply:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- For all values of `<regime>`, the instruction exists.
- For entries from the EL1&0 translation regime the instruction applies to cached copies of translation table entries with any VMID.
- For entries from a translation regime in which an ASID is valid, the instruction applies to all of the following cached copies of translation table entries from any lookup level in the translation table walk:
  - Global entries.
  - Non-global entries with any ASID.

R_{DNPQC} If `<type>` is `VMALL` in a TLB maintenance instruction, then all of the following apply:

- If and only if the value of `<regime>` is E1, then the instruction applies to the EL1&0 translation regime.
- The instruction applies to all cached copies of the stage 1 translation table entries from any lookup level in the translation table walk required to translate any address at the specified Exception level.
- For entries from the EL1&0 translation regime the instruction applies only to cached copies of translation table entries with the current VMID.
- For entries from a translation regime in which an ASID is valid, the instruction applies to all of the following cached copies of translation table entries from any lookup level in the translation table walk:
  - Global entries.
  - Non-global entries with any ASID.

R_{TBTPJ} If `<type>` is `VMALLS12` in a TLB maintenance instruction, then all of the following apply:

- If and only if the value of `<regime>` is E1, then the instruction applies to the EL1&0 translation regime.
- The instruction applies only to cached copies of translation table entries with the current VMID.
- For entries from a translation regime in which an ASID is valid, the instruction applies to all of the following cached copies of translation table entries from any lookup level in the translation table walk:
  - Global entries.
  - Non-global entries with any ASID.

R_{NMXFR} If `<type>` is `ASID` in a TLB maintenance instruction, then all of the following apply:

- If and only if the value of `<regime>` is E1, then the instruction applies to the EL1&0 translation regime when executing at EL1.
- The instruction applies to all cached copies of the stage 1 translation table entries from any lookup level in the translation table walk required to translate any address at the specified Exception level.
- The instruction applies only to non-global entries from the final lookup level that matches the specified ASID.

R_{WJWNN} If `<type>` is `VA` in a TLB maintenance instruction, then all of the following apply:

- For all values of `<regime>`, the instruction exists.
- The instruction applies to all cached copies of the stage 1 translation table entries from any lookup level in the translation table walk required to translate the address at the specified Exception level.
- For entries from a translation regime in which an ASID is valid, the instruction applies only to all of the following cached copies of translation table entries:
  - A non-global entry from any lookup level that matches the specified ASID.
  - A global entry from the final lookup level.

R_{PGHBM} If `<type>` is `VAL` in a TLB maintenance instruction, then all of the following apply:

- For all values of `<regime>`, the instruction exists.
- The instruction applies to all cached copies of the stage 1 translation table entries from the final lookup level in the translation table walk required to translate the address at the specified Exception level.
- The instruction applies only to cached copies of translation table entries from the final lookup level with the current VMID.
- For entries from a translation regime in which an ASID is valid, the instruction applies only to all of the following cached copies of translation table entries:
  - A non-global entry from the final lookup level that matches the specified ASID.
  - A global entry from the final lookup level.

R_{GKRKF} If `<type>` is `VAA` in a TLB maintenance instruction, then all of the following apply:

- If and only if the value of `<regime>` is E1, then the instruction applies to the EL1&0 translation regime when executing at EL1.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The instruction applies to all cached copies of the stage 1 translation table entries from any lookup level in the translation table walk required to translate the address at the specified Exception level.
- The instruction applies only to cached copies of translation table entries with the current VMID.
- The instruction applies to all of the following cached copies of translation table entries from any lookup level in the translation table walk:
  - Global entries.
  - Non-global entries with any ASID.

R_{CBRLG} If `<type>` is `VAAL` in a TLB maintenance instruction, then all of the following apply:

- If and only if the value of `<regime>` is E1, then the instruction applies to the EL1&0 translation regime when executing at EL1.
- The instruction applies to all cached copies of the stage 1 translation table entries from the final lookup level in the translation table walk required to translate the address at the specified Exception level.
- The instruction applies to all of the following cached copies of translation table entries from the final lookup level in the translation table walk:
  - Global entries.
  - Non-global entries with any ASID.

D5.12.6 Operation of the TLB maintenance instructions

R_{XQBCL} A TLB maintenance instruction applies whether a translation stage is enabled or disabled.

R_{KKDKL} A TLB maintenance instruction can affect any TLB entries that are not locked down.

I_{YGYRY} There is no guarantee that an unlocked TLB entry remains in the cache. Therefore, it is not possible to determine architecturally whether a TLB maintenance instruction has affected any TLB entries that were not specified by the instruction.

I_{LMGTM} The TLB maintenance instructions specify the Exception level of the translation regime to which they apply.

R_{TBLDD} A TLB maintenance instruction never generates an MMU abort.

I_{FNJXC} A TLB maintenance instruction is not affected by the current state of all of the following control bits involved in the translation process:

- In AArch64 state, all of the following:
  - SCTLR_EL1.M.
  - SCTLR_EL2.M.
  - HCR_EL2.{VM, RW}.
  - TCR_EL1.{TG1, EPD1, T1SZ, TG0, EPD0, T0SZ, AS, A1}.
  - TCR_EL2.{TG0, T0SZ}.
  - VTCR_EL2.{SL0, T0SZ}.
  - TTBR0_EL1.ASID.
  - TTBR1_EL1.ASID.

R_{FZTHY} Address-based TLB maintenance instructions are applied to the extent of the `<shareability>` specified by the instruction regardless of the Shareability assigned by any translation of the IA or Shareability attribute stored in TLB entries.

R_{LQLSD} When a TLB maintenance instruction is executed, all of the following apply:

- The minimum size of a TLB entry that is required to be invalidated from the TLB is at least the size specified by the corresponding translation table entry.
- The Contiguous bit does not affect the minimum size of a TLB entry that is required to be invalidated from the TLB.

I_{JQPNP} When invalidating a mapping greater than a translation granule size or block size, for example a mapping that is specified using the Contiguous bit, software is required to issue a TLB maintenance instruction for each block or granule within the configured contiguous region size to invalidate the entire mapping.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{MMZTY} An implementation is permitted to invalidate more than the minimum required TLB entries.

I_{ZWTPG} Arm recommends not invalidating entries that are not required to be invalidated to minimize the performance impact.

I_{RTRQB} Arm expects software to mostly use address-based TLB invalidation instructions that apply to entries cached from the last level of a stage 1 translation table walk to avoid unnecessary loss of the cached intermediate translation table entries.

D5.12.7 Broadcast TLB maintenance

R_{GYPWS} In all of the following cases, a broadcast TLB maintenance instruction is not required to perform any invalidation on the recipient PE within the target Shareability domain:

- A TLB maintenance instruction specifying a VA and affecting the EL1&0 translation regime is broadcast from a PE using a stage 1 translation granule size for that translation regime to a PE using EL1 and a different stage 1 translation granule size for that VA.
- A TLB maintenance instruction specifying a range of VAs or IPAs is broadcast from one PE to a PE that does not support the TLB range-based operations.

R_{YNPVV} The set of Requesters containing TLBs that support TLB range-based operations is defined by the system architecture.

I_{WHFDZ} Arm strongly recommends that within an Inner Shareable domain, all PEs support the same set of broadcast TLB range maintenance instructions.

R_{TVMHK} If an Armv7 PE is in the same Inner Shareable domain as an Armv8 PE, and the Armv8 PE issues a broadcast TLB maintenance instruction that is not defined in Armv7, then that instruction is not required to affect the TLBs of the Armv7 PE.

D5.12.8 Ordering and completion of TLB maintenance instructions

R_{TGVWD} For all of the following instructions, the effects of a TLB maintenance instruction can be observed in any order relative to that instruction unless a DSB is executed between the instructions:

- Any load or store instruction.
- Another TLB maintenance instruction.
- A data or instruction cache maintenance instruction.

R_{QRNJC} A TLB maintenance instruction executed by a PE causes a TLB maintenance operation to occur on each PE within the Shareability domain specified by the instruction.

R_{WWQZZ} When a TLB maintenance instruction is executed, *in-scope old translation information* is any translation information related to addresses within the scope of that instruction that is not consistent with one or more of the following:

- The architectural translation information held in the translation tables.
- Any architecture translation information that is Coherence-after the information held in the translation tables.

I_{YMDMJ} In-scope old translation information might be held in TLBs or other non-coherent caching structures.

R_{STPSQ} For a PE that handles a TLB maintenance operation the TLB maintenance operation is finished when all of the following apply:

- All memory accesses generated by that PE using in-scope old translation information are complete.
- All memory accesses, RWx, generated by that PE are complete, where RWx is the set of all memory accesses generated by instructions for that PE that appear in program order before an instruction I1 executed by that PE, where all of the following apply:
  - I1 uses the in-scope old translation information.
  - The use of the in-scope old translation information generates a synchronous Data Abort.
  - If I1 did not generate an abort from use of the in-scope old translation information, I1 would generate a memory access that RWx would be locally-ordered-before.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{DCJQM} When the TLB maintenance operations specified by a TLB maintenance instruction are finished for all PEs, the TLB maintenance instruction is complete.

R_{LXHLD} If a TLB maintenance instruction is complete, then no new memory accesses using the in-scope old translation information are architecturally performed by any observer that is affected by the TLB maintenance instruction.

I_{CPZSM} If it is impossible for software running on any observer to determine that the memory accesses have been performed, then speculative memory accesses can be performed using in-scope old translation information.

R_{PSMWS} When a DSB instruction is used to ensure the completion of TLB maintenance instructions, all of the following apply:

- The DSB is required to apply to both loads and stores.
- A DSB `NSH` is sufficient to ensure completion of TLB maintenance instructions that apply to a single PE.
- A DSB `ISH` is sufficient to ensure completion of TLB maintenance instructions that apply to PEs in the same Inner Shareable domain.

R_{YNJCP} When a PE, PEx, executes a TLB maintenance instruction, the instruction can complete at any time after it is issued, but the instruction is only guaranteed to be finished for a PE other than PEx after the execution of DSB by the PEx.

R_{RBWPQ} For the TLB of a PE that issues a TLB maintenance instruction, the instruction is guaranteed to be complete after the execution of DSB by that PE, followed by a Context synchronizatrion event.

I_{MGRRK} For more information, see Ordering requirements defined by the formal concurrency model.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.13 Caches

D5.13.1 Data and unified caches

R_{JHVQL} For data and unified caches, if all data accesses to an address do not use mismatched memory attributes, then the use of address translation is transparent to any data access to the address.
For more information, see Mismatched memory attributes.

R_{CTGML} For data accesses from the same observer to different VAs, if the accesses are translated to the same PA with the same memory attributes, then they are coherent and all of the following behaviors are guaranteed without requiring the use of barrier or cache maintenance instructions:

- Two writes to the same PA occur in program order.
- A read of a PA returns the value of the last successful write to that PA.
- A write to a PA that occurs, in program order, after a read of that PA, has no effect on the value returned by that read.

R_{NGPVX} When cache maintenance is done to a memory location, the effect of that cache maintenance is visible to all VA or IPA aliases of that physical memory location.

I_{FHPPX} The properties of data and unified caches are consistent with implementing the caches as physically-indexed, physically-tagged caches.

D5.13.2 Instruction caches

R_{GXBMD} An instruction cache has all of the following properties:

- An instruction cache is accessed only as a result of an instruction fetch.
- An instruction cache is never written to by a load or store instruction.
- Cache invalidation is the only cache maintenance that can be done on an instruction cache.

R_{XBHSB} The Arm architecture permits all of the following instruction cache implementations:

- *Physically-indexed, physically-tagged* (PIPT) instruction cache.
- *Virtually-indexed, physically-tagged* (VIPT) instruction cache.

I_{XSPPH} The CTR_EL0.L1Ip field identifies the form of the instruction caches.

R_{WNDVG} If CTR_EL0.DIC is 1, instruction cache maintenance is not required after overwriting instructions, including for different VA aliases of the affected Locations, regardless of the value of CTR_EL0.L1Ip.

R_{XLZJK} For all permitted instruction cache implementations, the Arm IVIPT Extension is also implemented, which reduces instruction cache maintenance requirements to requiring maintenance only after writing new data to a PA that holds an instruction.

I_{QNJPN} Previous versions of the Arm architecture permitted an instruction cache option that does not implement the Arm IVIPT Extension.

I_{GWSPL} For software to be portable between implementations that might use any of the permitted instruction cache implementations, that software is required to invalidate the instruction cache whenever any condition occurs that requires instruction cache maintenance on at least one of the instruction cache types.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.13.2.1 Physically-indexed, physically-tagged instruction caches

R_{YXNGL} For PIPT instruction caches, all of the following apply:

- If all instruction fetches to an address do not use mismatched memory attributes, then the use of address translation is transparent to any instruction fetch to the address.
- If instruction cache maintenance is done on a memory location, then the effect of that instruction cache maintenance is visible to all VA or IPA aliases of that physical memory location.

D5.13.2.2 Virtually-indexed, physically-tagged instruction caches

R_{LYZYY} For VIPT instruction caches, all of the following apply:

- If all instruction fetches to an address do not use mismatched memory attributes, then the use of address translation is transparent to any instruction fetch to the address.
- If instruction cache maintenance by address is done on a memory location, then the effect of that instruction cache maintenance is visible only to the VA supplied with the operation, and the effect of the invalidation might not be visible to any other VA or IPA aliases of that physical memory location.
- The only architecturally-guaranteed way to invalidate all VA or IPA aliases of a PA from a VIPT instruction cache is to invalidate the entire instruction cache.

D5.13.3 Cache maintenance requirements due to changing memory region attributes

I_{VPPPM} If changes are made to the memory region attributes in translation table entries, TLB maintenance might be required. For more information, see TLB maintenance.

I_{ZDQCY} The behaviors caused by mismatched memory attributes mean that if any of the following changes are made to the Inner Cacheability or Outer Cacheability attributes in translation table entries, then software is required to ensure that any cached copies of affected locations are removed from the caches, typically by cleaning and invalidating the locations from the cache levels that might hold copies of the locations affected by the attribute change:

- A change from Write-Back to Write-Through.
- A change from Write-Back to Non-cacheable.
- A change from Write-Through to Non-cacheable.
- A change from Write-Through to Write-Back.

For more information, see Mismatched memory attributes and Using break-before-make when updating translation table entries.

R_{ZNPYM} If the Shareability attributes in translation table entries are changed, then software is required to execute the following sequence to prevent possible coherency errors caused by mismatched memory attributes:

1. Make the memory location Non-cacheable, Outer Shareable.
2. Clean and invalidate the memory location from caches in the appropriate Shareability domain.
3. Change the Shareability attributes to the new values.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D5.14 Pseudocode description of VMSA address translation

I_{ZMHCV} The Arm architecture defines pseudocode that describes the translation table walk.

I_{STKYL} **Full Physical Address**

A complete PA necessary to identify a location in physical memory is captured by the type `FullAddress`. This is composed of:

- A bitstring `address`, which identifies the PA.
- An enumeration `paspace` which identifies the PA space.

I_{MXLCQ} **Address translation**

`AArch64.TranslateAddress()` acts as the entry point to VMSA and performs the required address translation based on the provided parameters and System register configurations. The function returns an `AddressDescriptor` structure holding valid data for either of the following:

- Target memory address and attributes for a non-faulting translation.
- Fault details holding data to be populated in syndrome registers.

I_{LTNYC} `AArch64.S1DisabledOutput()` assigns the appropriate memory attributes and flat maps the input address to the output address.

I_{KKVJD} **Translation table walk**

Stage 1 of translation has a separate walk function, `AArch64.S1Walk()`. Each use walk parameters extracted from related System registers. Parameters are collected based on the active translation regime.

I_{YMSFF} For a non-faulting walk, three items are returned by a translation table walk:

- The final walk state.
- The final descriptor fetched.
- The address of the final descriptor.

I_{KGLCH} The final descriptor and its address are used to update the descriptor as specified by Hardware management of the Access flag and Hardware management of the dirty state.

I_{RQKNH} A faulting walk could report one of the following at a specified level:

- Translation Fault.
- Address size Fault.
- Access flag Fault.

I_{DNSWC} **Hardware update of Translation Table descriptors**

The walk parameters collected from System registers indicate the ability to update the access flag or set write permissions within descriptors. This is controlled by the Dirty Bit Modifier, and the conditions specified in Hardware management of the Access flag and Hardware management of the dirty state. The translation function `AArch64.S1Translate()` set the appropriate descriptor bits returned by the walk functions and call `AArch64.MemSwapTableDesc()` to swap the old descriptor for the updated one in an atomic fashion.

I_{DKMRY} **Address decoding and calculation**

The walk state is initialized to hold the base address of the first translation table. For stage 1, the base address is acquired using `AArch64.S1TTBaseAddress()`, which decodes TTBR0_ELx and TTBR1_ELx registers depending on the regime and the range the VA corresponds to.

Indexing at the start level is shown in `AArch64.S1SLTTEntryAddress()` for stage 1. The final walk state would hold the base address for the output block or page; this is extracted from the Leaf descriptor in `AArch64.S1LeafBase()` for stage 1.

I_{GDTNP} **Memory attribute decoding**

If a stage of translation is enabled, fetched descriptors that are blocks or pages encode memory attributes assigned to the output of translation. Stage 1 memory attributes are decoded by the function `S1DecodeMemAttrs()`. The combination of stage 1 and stage 2 attributes by the function `S2CombineS1MemAttrs()` is presented to the MPU for the stage 2 translation. However, if FEAT_S2FWB is enabled, this behavior is overridden and memory attributes are decoded as specified in Stage 2 memory type and Cacheability attributes when FWB is enabled. This is captured by the function `AArch64.S2ApplyFWBMemAttrs()`.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{HGFRX} **Fault detection**

As soon as translation is invoked a reserve `FaultRecord` accompanies the process capturing the stage and level of translation as it proceeds. When a fault is detected, it is reflected in the `FaultRecord` and reported back as the result of translation with the most recent state to be reported already captured within. The following functions detect a certain type of fault:

- The `AArch64.S1CheckPermissions()` function checks permissions for stage 1.
- The `AArch64.S2HasAlignmentFaultDueToMemType()` functions detect an alignment fault for stage 1 and stage 2 respectively.
- The `AArch64.S1TxSZFaults()` function detects a Translation fault caused by erroneous configuration of TCR_ELx.T*n*SZ field.
- `AArch64.VAIsOutOfRange()` detects a stage 1 Translation fault caused by VAs larger than the address input size configured.

I_{PJBNQ} For more information, see function descriptions in aarch64/translation.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter D6
The AArch64 Protected Memory System Architecture

This chapter provides a system-level view of the Protected Memory System Architecture for any implementation that is compliant with the Armv8-R AArch64 architecture. It contains the following sections:

- About the Protected Memory System Architecture .
- Memory Protection Unit.
- Address translation regimes.
- Default memory map.
- MPU memory translations and faults.
- Protection region attributes and access permissions.
- MPU fault encodings.
- PMSAv8-64 implications for caches.
- Address tagging and pointer authentication support.
- Security model.
- Virtualization.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D6.1 About the Protected Memory System Architecture

R_{CCSZJ} The Armv8-R AArch64 architecture supports the *Protected Memory System Architecture* (PMSAv8-64) at EL1 and EL2.

I_{KFQTT} The PMSAv8-64 is based on *Memory Protection Units* (MPUs) that provide a memory protection scheme by defining *protection regions* in the address space.
The PMSAv8-64:

- Supports a unified memory protection scheme where an MPU manages instruction and data access. The MPUs defined by the architecture do not provide separate instruction protection regions and data protection regions in the address map.
- Defines MPU faults that are consistent with VMSAv8-64 fault definitions and reuses IFSC and DFSC fault encodings.
- Does not support virtual addressing and flat maps an input address to an output address.

D6.1.1 Protection regions

R_{LGFYG} The MPUs divide the address map into protection regions.

I_{MTLWF} A protection region is a contiguous memory region for which the MPU defines the memory attributes and the access permissions.

R_{DLWYC} Protection regions have all of the following properties:

- Are defined by a pair of registers, a Base Address Register, and a Limit Address Register for each Exception level:
  - PRBAR_ELx.
  - PRBAR<n>_ELx.
  - PRLAR_ELx.
  - PLBAR<n>_ELx.
- Have a minimum size of 64 bytes.
- Have a maximum size of the entire address map.
- Must not overlap.

R_{YNGFT} Memory attributes are provided by the Memory Attributes Indirection Registers:

- MAIR_ELx.
- AMAIR_ELx.

R_{QMXQS} The Memory Attribute Indirection Registers are indexed by the Base Address and Limit Address Registers to define the attributes for the Protection region.

I_{LFRRK} The definition of a protection region specifies the base and the limit of the region, the access permissions, and the memory attributes for the Protection region.

I_{GFSTP} If the MPU is disabled the PE uses an IMPLEMENTATION DEFINED default memory map. For more information see Default Memory Map.

D6.1.2 Address range

R_{RNZPF} The maximum supported address bit size is 48, or 52 if FEAT_LPA is enabled.

R_{HRWYX} A PE can choose to implement a smaller PA range and the actual implemented physical address range is defined by the ID_AA64MMFR0_EL1.PARange field.

R_{SVYSG} Any access to physical memory address outside the address range results in a memory fault.
For more information, see MPU memory translations and faults

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D6.2 Memory Protection Unit

I_{FQPST} An MPU checks whether the address used by a memory access matches all of the following:

- A memory protection region.
- The memory access permissions defined by the associated memory protection region.
- The memory attributes defined by the associated memory protection region.
- The Security state defined by the associated memory protection region.

R_{BQGDB} The PMSAv8-64 defines two MPUs:

EL1 MPU

The EL1 MPU can be configured from EL1 or EL2. The EL1 MPU controls the stage 1 of the Secure EL1&0 translation regime that defines the protection regions for accesses from EL1 and EL0.

EL2 MPU

The EL2 MPU can be configured only from EL2. The EL2 MPU controls:

- Stage 1 of the Secure EL2 translation regime that defines the protection regions for accesses from EL2.
- Stage 2 of the Secure EL1 &0 translation regime that defines the protection regions for accesses from EL1 and EL0.

Note

When HCR_EL2.VM is 1 and SCTLR_EL2.M is 1, then EL2 MPU modifies the access permissions and memory attributes that are assigned by the EL1 MPU.

R_{QLGWV} The PMSAv8-64 uses SCTLR_EL1.M to enable and disable the EL1 MPU. The EL1 MPU also supports a Background region, controlled by SCTLR_EL1.BR.

R_{GNWJH} The PMSAv8-64 uses SCTLR_EL2.M to enable and disable the EL2 MPU. The EL2 MPU also supports a Background region, controlled by SCTLR_EL2.BR.

For more information, see:

- Protection regions.
- Protection region attributes and access permissions.
- Default memory map.

D6.2.1 Default Cacheability

I_{CWZLJ} The PMSAv8-64 supports Default Cacheability for the stage 1 of the Secure EL1&0 translation regime access. For more information, see Memory region attributes.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D6.3 Address translation regimes

I_{WKVZH} In PMSAv8-64:

- Address translation describes the process of flat mapping the VA used by the PE, to the PA accessed in the memory system, and determining the access permissions and memory attributes of the target PA.
- A translation regime maps a VA to a PA using 1 or 2 stages of address translation to assign the access permissions and memory attributes of the target PA. When 2 translation stages are used, the intermediate address is called as an IPA.

R_{PPLTJ} The PMSAv8-64 architecture supports two translation regimes:

- Secure EL1&0 translation regime.
- Secure EL2 translation regime.

D6.3.1 Secure EL1&0 translation regime

R_{RSZBF} The Secure EL1&0 translation regime assigns the access permissions and memory attributes for any access from EL1 or EL0.

R_{JHPRR} This translation regime has 1 or 2 stages of translation:

- All accesses from EL1 or EL0 are translated by the EL1 MPU.
- When the value of HCR_EL2.VM is 1 and SCTLR_EL2.M is 1, the accesses are further translated by the EL2 MPU. This translation is a stage 2 translation, and can modify the access permissions and memory attributes that are assigned by the stage 1 translation.

R_{VHLGJ} For the EL1&0 stage 1 translation, an ADDRESS is in the protection region *n* if and only if:

- PRBAR<n>_EL1.BASE:'000000' <= ADDRESS <= PRLAR<n>_EL1.LIMIT:'111111'.

D6.3.2 Secure EL2 translation regime

R_{CTLZJ} The Secure EL2 translation regime assigns the access permissions and memory attributes for any access from EL2.

R_{CFQHD} This translation regime has a single stage of translation, stage 1, that is performed by the EL2 MPU.

R_{BWBJW} For the EL2 translation, an ADDRESS is in the protection region *n* if and only if:

- PRBAR<n>_EL2.BASE:'000000' <= ADDRESS <= PRLAR<n>_EL2.LIMIT:'111111'.

For more information, see Protection regions and Protection region attributes and access permissions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D6.4 Default memory map

R_{XVXND} The default memory map of the Armv8-R AArch64 architecture is IMPLEMENTATION DEFINED.

R_{ZTGTY} The memory attributes, access permissions, and Security state of the memory regions in the default memory map are IMPLEMENTATION DEFINED.

R_{LHZQT} The default memory map is same for EL1 and EL2 MPUs.

R_{PCHLR} For PMSAv8-64, the Background region is enabled and disabled using SCTLR_ELx.BR.

R_{ZQBYD} If the revelant MPU is disabled and the Background region is enabled (SCTLR_ELx.{BR,M} == {1,0}) then the default memory map is used to determine the memory attributes, access permissions and Security state for the Background region.

I_{WYVJN} The Armv8-R AArch64 architecture defines only the conditions required to access the default memory map, but not the memory map itself.

R_{SSMPX} Any access outside the implemented physical address range in the default memory map results in a fault.

R_{QVBFB} If the IMPLEMENTATION DEFINED default memory map is discontinuous, then the implementation must also define a generic permission and attribute to be used for memory access to all memory regions that are not covered by the default memory map. However, an implementation can also select a default memory map so that the accesses to these discontinuous memory regions, where no memory attributes are allocated, always result in memory faults.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D6.5 MPU memory translations and faults

I_{TBZYB} This section provides information on memory attributes and the MPU faults for the stage 1 Secure EL1&0, stage 1 Secure EL2, and the stage 2 Secure EL1&0 translation regimes.

For the EL1 MPU memory translations, this section describes the following:

- Table D6-1.
- Table D6-2.
- Table D6-3.
- Table D6-4.
- Figure D6-1.

For the EL2 MPU memory translations, this section describes the following:

- Table D6-5.
- Table D6-6.
- Table D6-7.
- Table D6-8.
- Table D6-9.
- Figure D6-2.
- Figure D6-3.

D6.5.1 EL1 MPU memory translations

I_{FTMQV} The EL1 MPU controls the stage 1 of the Secure EL1&0 translation regime.

R_{ZPLQC} Based on the values of HCR_EL2.DC and SCTLR_EL1.{M, BR}, the stage 1 of the Secure EL1&0 translation regime can have the following configurations for memory attributes, as described in Table D6-1.

Table D6-1 Stage 1 EL1&0 memory attributes

HCR_EL2	SCTLR_EL1		MPU hit	Memory attribute
DC	M	BR		
1	x	x	-	Default Cacheability
0	0	0	-	Accesses permitted
0	0	1	-	Default memory map
0	1	0	No	Not applicable, MPU Fault
0	1	0	Yes	MPU memory map
0	1	1	No	Default memory map
0	1	1	Yes	MPU memory map

R_{HFKTS} Based on the values of HCR_EL2.DC and SCTLR_EL1.{M, BR}, the stage 1 of the Secure EL1&0 translation regime can result in MPU faults for EL1 accesses, as described in Table D6-2.

Table D6-2 Stage 1 MPU faults for EL1 access

HCR_EL2	SCTLR_EL1		MPU hit	MPU faults
DC	M	BR		
1	x	x	-	No Fault or Address size fault
0	0	0	-	No Fault or Address size fault

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

HCR_EL2	SCTLR_EL1		MPU hit	MPU faults
DC	M	BR		
0	0	1	-	No Fault, or Background region Translation fault, or Background region Permission fault
0	1	0	No	Translation fault
0	1	0	Yes	No Fault or Permission fault
0	1	1	No	No Fault, or Background region Translation fault, or Background region Permission fault
0	1	1	Yes	No Fault or Permission fault

R_{ZSPZK} Based on the values of HCR_EL2.DC and SCTLR_EL1.{M, BR}, the stage 1 of the Secure EL1&0 translation regime can result in MPU faults for EL0 accesses, as described in Table D6-3.

Table D6-3 Stage 1 MPU faults for EL0 access

HCR_EL2	SCTLR_EL1		MPU hit	MPU faults
DC	M	BR		
1	x	x	-	No Fault or Address size fault
0	0	0	-	No Fault or Address size fault
0	0	1	-	No Fault, or Background region Translation fault, or Background region Permission fault
0	1	x	No	Translation fault
0	1	x	Yes	No Fault or Permission fault

R_{QMPWZ} If HCR_EL2.{DC, TGE} is not {0, 0}, then the PE behaves as if the value of the SCTLR_EL1.BR is 0 for all purposes other than returning the value of a direct read of the field.

D6.5.2 EL1 MPU faults

I_{RBZDF} Each EL1 MPU protection region is defined using the PRBAR_EL1 and PRLAR_EL1 registers.

R_{YFYJN} The MPU checks the input address with each protection region, and an address is considered to match a region if:

```
Address >= PRBAR_EL1.BASE:'000000' && Address <= PRLAR_EL1.LIMIT:'111111'
```

R_{HTZMK} Based on MPU protection region checks, the EL1 MPU can raise the following responses as described in Table D6-4.

Table D6-4 EL1 MPU fault types

Protection region match	Permission	MPU response
No match	-	Translation fault
Multiple	-	Translation fault
Single	Denied	Permission fault
	Allowed	Valid

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{DFBZT} If the EL1 MPU is disabled and the input address is larger than the implemented PA size, then a level 0 address size fault is generated.

R_{GSYFG} If the EL1 MPU is enabled and the input address is larger than the implemented PA size, then a level 0 translation fault is generated.

R_{GHZRY} Transactions permitted by the EL1 MPU are presented to stage 2 permission checks by the EL2 MPU.

I_{JBNSP} Depending on the configuration in the PMSAv8-64 registers, the memory attributes of an address can be defined by an MPU protection region or a Background region.

D6.5.3 MPU fault check for the stage 1 Secure EL1&0 translation

I_{TSMHY} Figure D6-1 shows the MPU fault check sequence for the stage 1 of the Secure EL1&0 translation regime.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

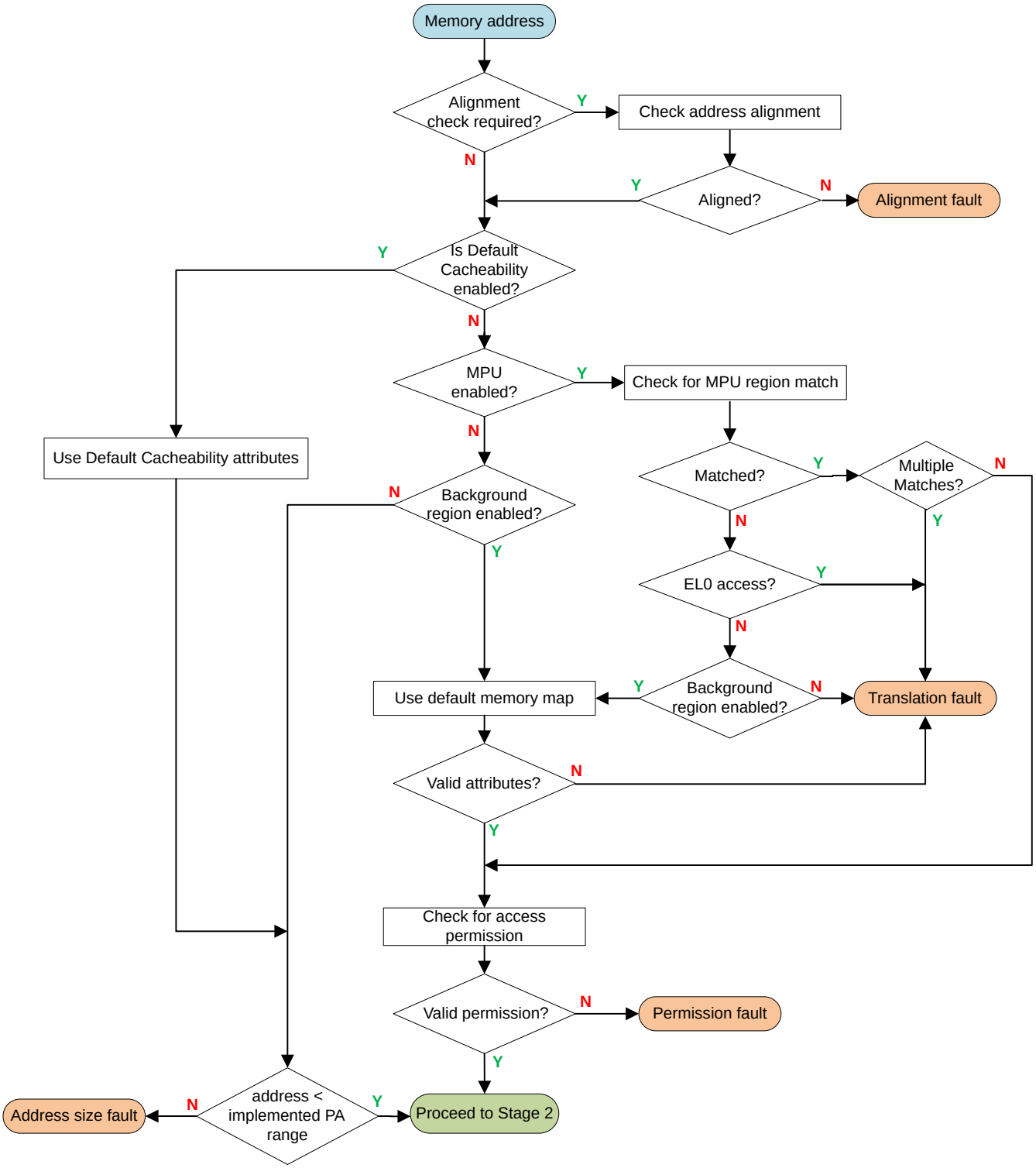


Figure D6-1 MPU fault check sequence for the stage 1 Secure EL1&0 translation

D6.5.4 EL2 MPU memory translations

R_{FJDCZ} Based on the values of SCTLR_EL2.{M, BR}, the stage 1 of the Secure EL2 translation regime can have the following configurations for memory attributes, as described in Table D6-5.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D6-5 Stage 1 EL2 memory attributes

SCTLR_EL2		MPU hit	Memory attributes
M	BR		
0	0	-	Accesses permitted
0	1	-	Default memory map
1	0	No	Not applicable. MPU Fault.
1	0	Yes	MPU memory map
1	1	No	Default memory map
1	1	Yes	MPU memory map

Table D6-6 lists the configurations for stage 1 MPU faults for EL2 access.

Table D6-6 Stage 1 MPU faults for EL2 access

SCTLR_EL2		MPU hit	MPU faults
M	BR		
0	0	-	No Fault or Address size fault
0	1	-	No Fault, or Background region Translation fault, or Background region Permission fault
1	0	No	Translation fault
1	0	Yes	No Fault or Permission fault
1	1	No	No Fault, or Background region Translation fault, or Background region Permission fault
1	1	Yes	No Fault or Permission fault

R_{CZDBJ} Based on HCR_EL2.VM and SCTLR_EL2.{M, BR}, the stage 2 of the Secure EL1&0 translation regime can have the following configurations for memory attributes, as described in Table D6-7.

Table D6-7 Stage 2 EL1&0 memory attributes

HCR_EL2	SCTLR_EL2		MPU hit	Memory attribute
VM	M	BR		
0	x	x	-	Stage 2 translation disabled
1	0	0	-	CONSTRAINED UNPREDICTABLE
1	0	1	-	Default memory map
1	1	x	No	Not applicable. MPU Fault.
1	1	x	Yes	MPU memory map

Table D6-8 lists the configurations for MPU faults for the stage 2 of the Secure EL1&0 translation regime.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D6-8 Stage 2 EL1&0 MPU faults

HCR_EL2	SCTLR_EL2		MPU hit	MPU faults
VM	M	BR		
0	x	x	-	No Fault
1	0	0	-	CONSTRAINED UNPREDICTABLE
1	0	1	-	No Fault, or Background region Translation fault, or Background region Permission fault, or Translation fault due to Secure Check, if enabled.
1	1	x	No	Translation fault
1	1	x	Yes	No Fault, or Permission fault, or Translation fault due to Secure Check, if enabled.

R_{XTWMV} There are no separate configurations for the protection regions for the stage 1 of the Secure EL2 and the stage 2 of the Secure EL1&0 translations. Memory accesses for both translations are controlled by the same MPU configuration registers, PRBAR_EL2 and PRLAR_EL2.

Note

In Armv8-A, there are separate translation table base registers for the stage 1 of the Secure EL2 and the stage 2 of the Secure EL1&0 translation regimes.

D6.5.5 EL2 MPU faults

I_{QVJVG} Each EL2 MPU protection region is defined using PRBAR_EL2 and PRLAR_EL2.

R_{YHTXR} The MPU checks the input address with each protection region and an address is considered to match a region if:

```
Address >= PRBAR_EL2.BASE:'000000' && Address <= PRLAR_EL2.LIMIT:'111111'
```

R_{FXSQK} Based on MPU protection region checks, the EL2 MPU can raise the following responses as described in Table D6-9.

Table D6-9 EL2 MPU fault types

Protection region match	Permission	MPU response
No match	-	Translation fault
Multiple	-	Translation fault
Single	Denied	Permission fault
Single	Allowed	Valid

R_{JGXXQ} If the EL2 MPU is disabled for the stage 1 of the Secure EL2 translation regime, then any access to an address outside the implemented PA range raises a level 0 address size fault.

R_{SLNZH} If the EL2 MPU is enabled for the stage 1 of the Secure EL2 translation regime, and the input address is larger than the implemented PA range, then a level 0 translation fault is generated.

D6.5.6 MPU fault check for the stage 1 Secure EL2 translation

I_{BMMTX} Figure D6-2 shows the MPU fault check sequence for the stage 1 of the Secure EL2 translation regime.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

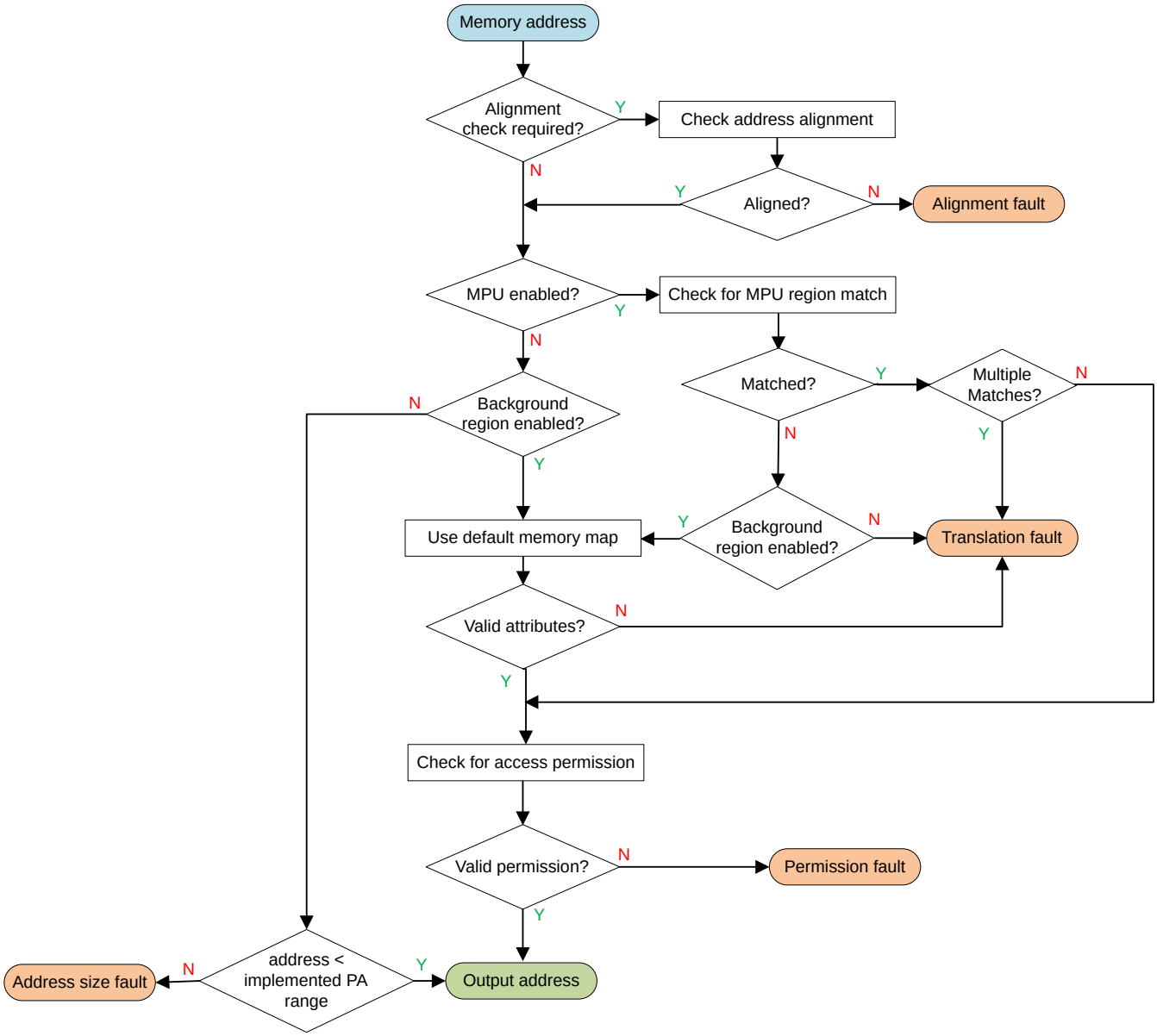


Figure D6-2 MPU fault check sequence for the stage 1 Secure EL2 translation

D6.5.7 MPU fault check for the stage 2 Secure EL1&0 translation

I_{DTXVK} Figure D6-3 shows the MPU fault check sequence for the stage 2 of the Secure EL1&0 translation regime.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

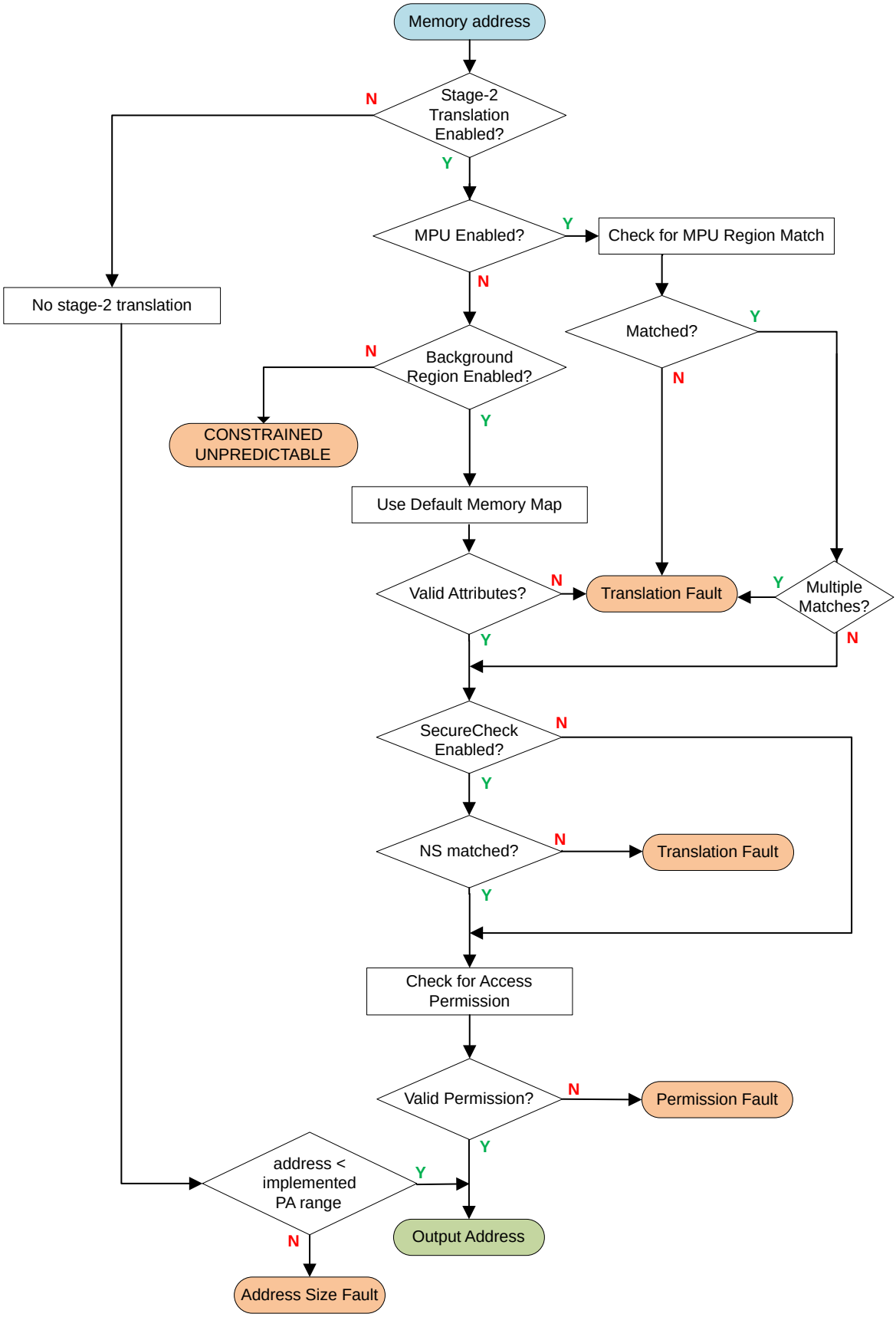


Figure D6-3 MPU fault check sequence for the stage 2 Secure EL1&0 translation

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D6.6 Protection region attributes and access permissions

R_{HJQLL} The protection region attribute fields control the Memory type, Cacheability, and Shareability attributes of the region. PMSAv8-64 uses the same memory types and memory attributes as VMSAv8-64.

R_{ZDXDJ} The memory attributes and access permissions for a protection region are defined by:

- The PRBAR_EL1 and PRLAR_EL1 registers, or the PRBAR_EL2 and PRLAR_EL2 registers that define the protection region.
- The MAIR_EL1.Attr<*n*> or MAIR_EL2.Attr<*n*> field that is indexed by PRLAR_EL1.AttrIndx or PRLAR_EL2.AttrIndx, respectively.

For the Secure EL1&0 translation regime, when HCR_EL2.VM is 1, the stage 1 memory attribute and access permission assignments are combined with the stage 2 assignments, as described in Combining memory attributes and access permissions.

For more information, see Memory attributes and access permission mappings.

D6.6.1 Protection region attributes

R_{MJDVK} The PMSAv8-64 uses the same memory attributes defined by the VMSAv8-64, and the MPU enables configuration of the attributes for each protection region using MAIR_EL1 and MAIR_EL2 for the stage 1 EL1&0 and EL2 translations.

R_{NJQMD} The PMSAv8-64 also enables configuration of each protection region to map to Secure or Non-secure address space using the NS bit in the EL1 and EL2 MPU configuration registers.

R_{GRFHL} Writes to MPU registers are only guaranteed to be visible following a *Context synchronization event* and DSB operation.

R_{ZYCWG} If there are multiple protection regions allocated to the same coherency granule, then PMSAv8-64 follows the mismatched memory attributes rules to access any byte within that coherency granule.

R_{VMRLQ} If the Security states of protection regions are different, then implementation must ensure that these regions are not allocated to the same coherency granule. For the stage 2 EL1&0 translations, memory attributes encoding in the MAIR_EL2 register is defined in Table D6-10.

Table D6-10 Meaning of Attr[7:4]

Attr [7:6]	Attr [5:4]	Memory type	Outer cache policy
00	00	Device memory	Not Applicable (NA)
00	!=00	Normal memory	Write-Through (WT)
01	00	Normal memory	Non Cacheable (NC)
01	!=00	Normal memory	Write-Back (WB)
10	xx	Normal memory	Write-Through (WT)
11	xx	Normal memory	Write-Back (WB)

R_{BQYGF} When Attr[7:4] is 0b0000, Attr[3:0] defines the type of Device memory. In this case, Attr[1:0] != 0b00 gives UNPREDICTABLE behavior as defined by Armv8-A. Table D6-11 describes the meaning of Attr<3.0> when Attr[7:4] is 0b0000.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D6-11 Attr<3.0> Meaning when Attr[7:4] is `0b0000`

Attr [3:2]	Attr [1:0]	Memory type
`00`	`00`	Device-nGnRnE
`01`	`00`	Device-nGnRE
`10`	`00`	Device-nGRE
`11`	`00`	Device-GRE

R_{TMGLY} When Attr[7:4] is not `0b0000`, Attr[3:0] defines the Inner Cache Policy, and Attr[3:0] = `0b0000` gives UNPREDICTABLE behavior as defined by Armv8-A. Table D6-12 describes the meaning of Attr<3.0> when Attr[7:4] is not `0b0000`.

Table D6-12 Attr<3.0> Meaning when Attr[7:4] is not `0b0000`

Attr [3:2]	Attr [1:0]	Memory type	Inner Cache Policy
`00`	!=`00`	Normal memory	Write-Through (WT)
`01`	`00`	Normal memory	Non Cacheable (NC)
`01`	!=`00`	Normal memory	Write-Back (WB)
`10`	`xx`	Normal memory	Write-Through (WT)
`11`	`xx`	Normal memory	Write-Back (WB)

For more information, see The AArch64 Virtual Memory System Architecture.

D6.6.2 Access control

I_{VXZDW} The Base and Limit registers in the PMSAv8-64 contain the same access permission and security control bits present in the VMSAv8-64 translation table desciptors.

R_{MCJFK} PMSAv8-64 also does not support Privileged execute-never (PXN) and Unprivileged execute-never (UXN) bits. PMSAv8-64 behaves as if PXN = XN and UXN = XN, and follows the same rule defined by VMSAv8-64 in combining permission attributes.

R_{XHFGG} The following table lists the stage 1 permissions and the access type that is permitted when that permission is present. If the stage 1 permission is not present, that access type is not permitted.

Table D6-13 Stage 1 permissions

Stage 1 permission	Access type permitted
UnprivRead	Unprivileged data read in the EL1&0 translation regime.
UnprivWrite	Unprivileged data write in the EL1&0 translation regime.
PrivRead	Privileged data read in the EL1&0, and EL2 translation regimes.
PrivWrite	Privileged data write in the EL1&0, and EL2 translation regimes.
Execute	Instruction execution in the EL1&0, and EL2 translation regimes.

R_{PBTCW} The following table shows the possible data access permissions at EL0 and the corresponding higher Exception level.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D6-14 Data access permissions for a stage 1 translation supporting two Exception levels

AP[2:1]	Permissions
00	PrivRead, PrivWrite
01	PrivRead, PrivWrite, UnprivRead, UnprivWrite
10	PrivRead
11	PrivRead, UnprivRead

I_{HKLTT} All of the following controls can ensure that memory locations are not both writable and executable:

- For privileged accesses in all translation regimes, PrivWXN.
- For unprivileged accesses in the EL1&0 translation regime, UnprivWXN.

I_{TNLPM} In an MPU controlled memory access that applies to EL0 and a higher Exception level, then the following table shows the possible stage 1 memory access permissions, using all of the following notations:

- XN is in the applicable PRBAR<n>_ELx register at the higher Exception level to which the translation regime applies.
- AP[2:1] is the Effective value of AP[2:1].
- WXN is in the SCTLR_ELx register at the higher Exception level to which the translation regime applies.

Table D6-15 Summary of possible memory access permissions using Direct permissions for a stage 1 translation supporting two Exception levels

XN	AP[2:1]	WXN	Permission
0	00	0	PrivRead, PrivWrite, Execute
0	00	1	PrivRead, PrivWrite, PrivWXN, Execute
0	01	0	PrivRead, PrivWrite, UnprivRead, UnprivWrite, Execute
0	01	1	PrivRead, PrivWrite, UnprivRead, UnprivWrite, UnprivWXN
0	10	x	PrivRead, Execute
0	11	x	PrivRead, Execute, UnprivRead, Execute
0	00	x	PrivRead, PrivWrite, Execute
1	00	0	PrivRead, PrivWrite,
1	00	1	PrivRead, PrivWrite, PrivWXN
1	01	x	PrivRead, PrivWrite, UnprivRead, UnprivWrite
1	10	x	PrivRead
1	11	x	PrivRead, UnprivRead
1	00	x	PrivRead, PrivWrite
1	01	x	PrivRead, PrivWrite, UnprivRead, UnprivWrite
1	10	x	PrivRead
1	11	x	PrivRead, UnprivRead

R_{JCKXP} If the values of SCTLR_EL1.{WXN, M} is {1,1} then in the in EL1&0 translation regime a memory region that is marked as PrivWrite will be Execute-never at EL1 and if marked as UnprivWrite will be Execute-never at EL0.

R_{NJNYR} If the value of SCTLR_EL1.{M, BR} is {0, 1}, then for the Secure EL1&0 translation regime, any memory region that is writable at EL0, is also executable from EL1 if that address is marked as executable by the Background region.

R_{SCNQL} Access granted in the stage 1 of the Secure EL1&0 translation regime by the EL1 MPU is subject to further qualification by the EL2 MPU in the stage 2 of the Secure EL1&0 translation regime.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{FZDRR} For selecting memory attributes and defining access permissions, PMSAv8-64 follows the same prioritization rules defined by VMSAv8-64.

R_{PCXZQ} PMSAv8-64 does not support hierarchical control bits defined in the VMSAv8-64 table descriptors.

R_{PVRXB} Armv8-R AArch64 supports FEAT_PAN. If the value of SCTLR_EL1.M is 1, FEAT_PAN is applied to the stage 1 of the Secure EL1&0 translation regime.

D6.6.3 Memory attributes and access permission mappings

R_{LXGMJ} Memory attributes, Security states, and access permission information present in the translation table descriptors of VMSAv8-64 are mapped to MPU configuration registers in PMSAv8-64, as described in Table D6-16.

Table D6-16 Memory attributes and access permission mappings

Attributes and permissions	MPU register fields	Description
Non-secure (NS)	PRLAR_EL1.NS PRLAR_EL2.NS	Specifies whether the translated address is in the Secure or Non-secure address space.
Access Permission (AP)	PRBAR_EL1.AP PRBAR_EL2.AP	Defines the Access permissions for the protection region.
Execute Never (XN)	PRBAR_EL1.XN PRBAR_EL2.XN	Defines the Execute-never attribute for the protection region.
Shareability (SH)	PRBAR_EL1.SH PRBAR_EL2.SH	Defines the Shareability for a Normal memory region. For any type of Device memory or Normal Non-cacheable memory, the value of the SH[1:0] field is IGNORED.
Attribute Index (AttrIndx)	PRLAR_EL1.AttrIndx PRLAR_EL2.AttrIndx	Indexes an Attr<n> field in the MAIR_EL1 or MAIR_EL2 register, which gives the memory type and memory attributes.

D6.6.4 Combining memory attributes and access permissions

I_{YPVNH} The PMSAv8-64 uses the same architecture rules as Armv8-A for combining stage 1 and stage 2 memory attributes and the access permissions for the Secure EL1&0 translation regime.

For more information see Combining stage 1 and stage 2 memory type attributes.

D6.6.4.1 Stage 2 forced Write-Back Feature

R_{BTHTX} If FEAT_S2FWB is enabled (HCR_EL2.FWB=1), the Inner and Outer Memory attributes for the stage 2 of the Secure EL1&0 translation regime must be the same with the same encoding, otherwise the combined attribute is UNKNOWN.

- If FEAT_S2FWB is enabled and the memory attributes for the stage 2 of the Secure EL1&0 translation regime are Write-Back (WB) with Attr[7:6] = 0b11, then the combined attribute is WB. If the memory attribute is assigned by the MPU, the attribute encoding is derived from MAIR_EL2.Attr[7:6]. If the memory attribute is assigned from Background region, then the encoding is derived from the memory region configuration in the Background region.
- If FEAT_S2FWB is not enabled, then stage 1 and stage 2 memory attributes of the Secure EL1&0 translation regime are combined using Armv8-A rule for combining memory attributes when FEAT_S2FWB is not enabled.

R_{CWSCK} In Armv8-A, the memory attribute encoding that enables FEAT_S2FWB is MemAttr[4:2] = 0b110 in stage 2 block or page descriptor of the Secure EL1&0 translation regime, while in Armv8-R AArch64, it is MAIR_EL2.Attr[7:6] = 0b11. Therefore, FEAT_S2FWB architecture rules defined for the 0b110 encoding in Armv8-A must be applied to the 0b11 encoding in Armv8-R AArch64.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{RWXST} For more information, see Stage 2 memory type and Cacheability attributes when FEAT_S2FWB is enabled.

D6.6.5 Enabling and disabling the caching of memory accesses

I_{TLWNZ} The Armv8-R AArch64 architecture follows the same rule as the Armv8-A AArch64 architecture for enabling and disabling cache, and can force all memory locations with the Normal memory type to be treated as Non-cacheable regardless of their assigned Cacheability attribute using SCTLR_ELx.{I,C} and HCR_EL2.{ID,CD}.
For more information, see Cacheability attributes for Normal memory.

R_{HDGGS} For a PMSAv8-64 based address translation, Armv8-R AArch64 extends the functionality of SCTLR_ELx.{I,C} and HCR_EL2.{ID,CD} for accesses to Background region also.

D6.6.6 Enabling and disabling stages in translation regimes

R_{FQGZD} The EL1 MPU controls the stage 1 of the Secure EL1&0 translation regime.

R_{GGQSJ} The EL2 MPU controls the stage 2 of the Secure EL1&0 translation regime and the stage 1 of the Secure EL2 translation regime.

R_{ZNDLM} For the Secure EL1&0 translation regime:

- If SCTLR_EL1.{M, BR} is {0, 0}, then stage 1 translation is disabled.
- If HCR_EL2.VM is 0, then stage 2 translation is disabled.
- If HCR_EL2.VM is 1 and SCTLR_EL2.{M, BR} is {0, 0}, then the behavior is CONSTRAINED UNPREDICTABLE with the following permitted values:
  - The stage 2 memory attribute becomes UNKNOWN.
  - Raise stage 2 level 0 translation fault.

R_{RRRVB} For the Secure EL2 translation regime if SCTLR_EL2.{M, BR} is {0, 0}, then translation is disabled.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D6.7 MPU fault encodings

R_{VSGFD} PMSAv8-64 defines MPU faults that are consistent with the VMSAv8-64 fault definitions and reuses the ELR_ELx.{IFSC, DFSC} fault encodings.

R_{WKWLY} Each MPU raises faults on invalid memory accesses, including accesses to memory regions outside address ranges and the accesses without sufficient permissions. Table D6-17 describes the MPU fault encodings mapped from VMSAv8-64.

Table D6-17 PMSAv8-64 fault encodings

Memory faults	MPU fault encodings mapped from VMSAv8-64	Description
Alignment fault	Alignment fault	Unaligned memory access.
Translation fault	Translation fault, level 0	Invalid input address. There is no valid mapping or valid memory attributes for the input address.
Permission fault	Permission fault, level 0 (`0b001100`)	Insufficient access permissions.
Address size fault	Address size fault, level 0	Generated output address is out of range of the implemented physical address.
Access flag fault		Not applicable for PMSAv8-64.
TLB conflict abort		Not applicable for PMSAv8-64.
Synchronous abort (translation table walk)		Not applicable for PMSAv8-64.

For more information, see The AArch64 System Level Programmers' Model.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D6.8 PMSAv8-64 implications for caches

I_{KJCWD} Enabling, reconfiguring, or reprogramming any memory region can result in new and different memory attributes for a previously accessed or speculatively accessed address. In this case, the rules for mismatched memory attributes apply.

See Mismatched memory attributes for more information.

D6.8.1 Cache line length

R_{RLGGC} The PMSAv8-64 permits the definition of memory regions that might be smaller than a cache line in the implementation. Therefore, the following rules apply:

- If the MPU is configured so that multiple differing attributes apply to a single cache line, then for any access to that cache line, the rules for mismatched memory attributes apply.
- Marking any part of a cache line as Write-Back permits the entire line to be treated as Write-Back.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D6.9 Address tagging and pointer authentication support

R_{QSMRT} The Armv8-R AArch64 architecture supports Address tagging.

R_{YGVMZ} The Armv8-R AArch64 architecture supports FEAT_PAuth2 feature that is defined in the Armv8-A architecture with a modified definition of PAC field as described below:

- When Address tagging is used, the PAC field is Xn[54:bottom_PAC_bit].
- When Address tagging is not used, the PAC field is Xn[63:56, 54:bottom_PAC_bit].

R_{FJJMS} In PMSAv8-64, the bottom_PAC_bit is the maximum physical address size as indicated by ID_AA64MMFR0_EL1.PARange.

For more information, see:

- Pointer authentication.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D6.10 Security model

R_{JHBHB} The Armv8-R AArch64 architecture always executes code in Secure state with EL2 as the highest Exception level and:

- It is not possible to switch to Non-secure state.
- EL0, EL1, and EL2, all run in Secure state.

D6.10.1 Secure EL2

I_{DPDGZ} The Armv8-R AArch64 architecture adopts FEAT_SEL2 feature from Armv8.4-A architecture extension and introduces it as PMSAv8-64 based architecture feature.

I_{SBSHY} The protection region configuration registers for EL1 and EL2 MPUs, PRLAR_EL1 and PRLAR_EL2, are extended to include the NS bit that has the same behavior as NS bit in the translation table descriptor of VMSAv8-64.

R_{XNFXC} Security controls for translation table walks are not supported in PMSAv8-64.

R_{BJMVG} The NS bit in the PRLAR_EL1 and PRLAR_EL2 specifies whether the output address is in the Secure or Non-secure address space. Each protection region can be independently configured to the Secure or Non-secure address space.

R_{PTBZB} The configuration bits required to implement Secure and Non-secure states in VMSAv8-64 are mapped to PMSAv8-64 as described in Table D6-18.

Table D6-18 VMSAv8-64 and PMSAv8-64 security configuration

VMSAv8-64	PMSAv8-64
NS bit in translation table	PRLAR_EL1.NS/PRLAR_EL2.NS
NSTable bit in translation table	Not Applicable
VSTCR_EL2.SW	Not Applicable
VSTCR_EL2.SA	VSTCR_EL2.SA
VTCR_EL2.NSW	Not Applicable
VTCR_EL2.NSA	VTCR_EL2.NSA

For more information, see:

- Address translation.
- Translation table descriptor formats.
- Security state of translation table lookups.
- Translation process.

D6.10.1.1 Secure EL1&0 translation

R_{PDXNY} As the stage 2 of the Secure EL1&0 translation regime is controlled by the EL2 MPU and there is no page-based address translation or translation table walk, the following modifications are added to FEAT_SEL2:

- VTCR_EL2.NSW and VSTCR_EL2.SW control bits are RES0.
- VSTTBR_EL2 register is not present.
- Adds a secure check control, VSTCR_EL2.SC.

R_{ZXXGF} VSTCR_EL2.SA and VTCR_EL2.NSA controls are supported and have the same functionality as FEAT_SEL2 but are applied to PA in the Secure EL1&0 translation regime.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{WXGLD} VSTCR_EL2.SC has no effect on the Secure EL2 translation regime. Table D6-19 describes the behavior of VSTCR_EL2.SC for the Secure EL1&0 translation regime.

Table D6-19 Secure check behavior in the Secure EL1&0 translation regime

VA	EL1 MPU NS attribute	IPA	EL2 MPU NS attribute	SC	PA
Secure	0	Secure	0	x	Secure
Secure	1	Non-secure	1	x	Non-secure
Secure	0	Secure	1	0	Non-secure
Secure	0	Secure	1	1	Fault
Secure	1	Non-secure	0	0	Non-secure
Secure	1	Non-secure	0	1	Fault

R_{TYJWG} EL1 and EL0 access is further subjected to VSTCR_EL2.SA and VTCR_EL2.NSA controls as defined by Armv8.4-A. Table D6-20 describes the behavior of VSTCR_EL2.SA and VTCR_EL2.NSA in the Secure EL1&0 translation regime.

Table D6-20 VSTCR_EL2.SA and VTCR_EL2.NSA behavior in the Secure EL1&0 translation regime

PA.NS	VSTCR_EL2.SA	VTCR_EL2.NSA	PA.NS (final)
0	0	x	0
0	1	Behaves as 1	1
1	0	0	0
1	0	1	1
1	1	x	1

R_{TBFFN} The Armv8-R AArch64 architecture follows the Armv8-A architecture rules on whether VSTCR_EL2.SA and VTCR_EL2.NSA controls should be applied.

- If HCR_EL2.VM=1 and SCTLR_EL2.M=1 or SCTLR_EL2.BR=1, then stage 2 Secure EL1&0 translation is enabled.
- If HCR_EL2.VM=0, then stage 2 Secure EL1&0 translation is disabled.

For more information, see The AArch64 System Level Programmers' Model.

D6.10.2 Secure EL2 translation

R_{JNLSG} The Armv8-R AArch64 architecture implements PMSAv8-64 at stage 1 of the Secure EL2 translation regime. Therefore, there is no translation table walk and address mapping.

R_{JPBSD} The NS bit in the EL2 MPU protection region register determines whether the output address must be looked for in either the Secure or Non-secure address space.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D6.11 Virtualization

R_{SYWRQ} Armv8-R AArch64 implements a permission-based containerization by introducing a stage 2 translation.

I_{XTHJL} The stage 2 translation uses an MPU controlled by a hypervisor running at EL2.

R_{PPJMB} The MPU does not perform address mapping and only checks permissions.

I_{NBZNG} Therefore, Armv8-R AArch64 relies on hypervisor configured permission attributes of the memory region to implement containerization.
For more information, see The AArch64 System Level Programmers' Model.

D6.11.1 Support for Guest operating systems

R_{LHPVJ} The hypervisor supports Guest operating systems using PMSAv8-64 on a per guest basis.

R_{CWHFT} Memory accesses by PMSAv8-64 guests are further validated by the EL2 MPU, controlled by the hypervisor.

R_{JNZKY} PMSAv8-64 guests have access to the EL1 MPU for isolating individual tasks.

I_{SZSQD} If multiple PMSAv8-64 guests are present, then these guests must be configured to use non-conflicting physical memory addresses. Virtualization is supported by the EL2 MPU at stage 2 of the Secure EL1&0 translation regime.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter D7
The Generic Timer in AArch64 state

This chapter describes the implementation of the Arm Generic Timer. It includes an overview of the AArch64 System register interface to an Arm Generic Timer.

It contains the following sections:

- About the Generic Timer.
- The AArch64 view of the Generic Timer.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D7.1 About the Generic Timer

Figure D7-1 shows an example System on Chip that uses the Generic Timer as a system timer. In this figure:

- This manual defines the architecture of the individual PEs in the multiprocessor blocks.
- The *Arm Generic Interrupt Controller Architecture Specification* defines a possible architecture for the interrupt controllers.
- Generic Timer functionality is distributed across multiple components.

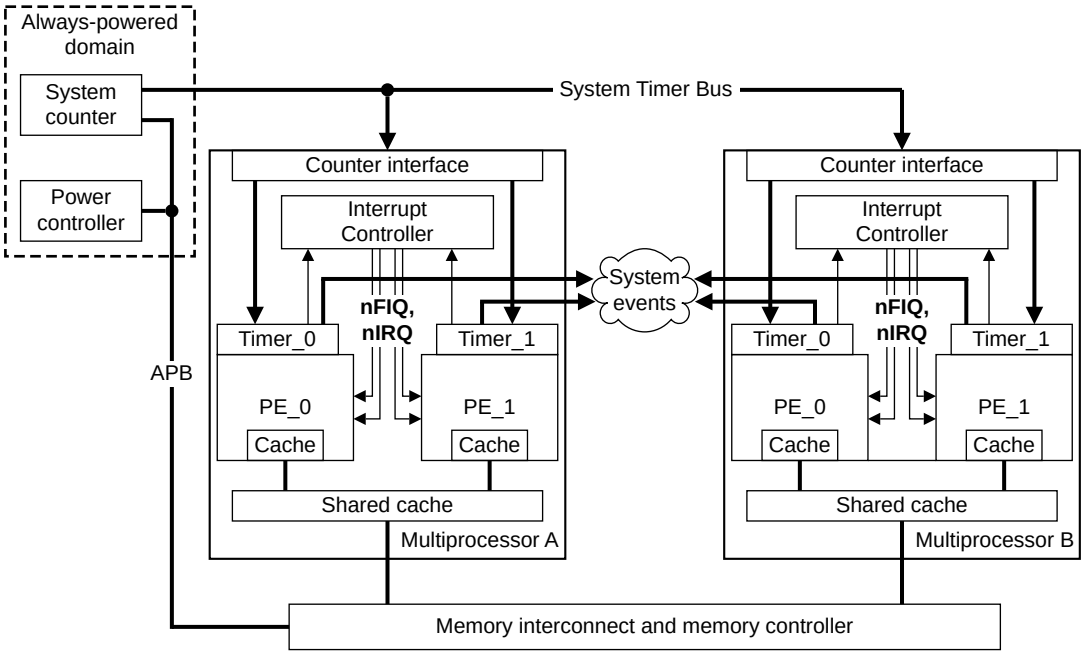


Figure D7-1 Generic Timer example

The Generic Timer:

- Provides a system counter that measures the passing of time in real-time.

  Note

  The Generic Timer can also provide other components at a system level, but Figure D7-1 does not show any such components.

- Supports *virtual counters* that measure the passing of virtual-time. That is, a virtual counter can measure the passing of time on a particular virtual machine.
- Can trigger events after a period of time has passed. The timers:
  - Can be used as count-up or as count-down timers.
  - Can operate in real-time or in virtual-time.

This chapter describes an instance of the Generic Timer component that Figure D7-1 shows as Timer_0 or Timer_1 within the Multiprocessor A or Multiprocessor B block.

A Generic Timer implementation must also include a memory-mapped system component. This component:

- Must provide the System counter shown in Figure D7-1.
- Optionally, can provide timer components for use at a system level.

System Level Implementation of the Generic Timer describes this memory-mapped component.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D7.1.1 The full set of Generic Timer components

Within a system that might include multiple PEs, a full set of Generic Timer components is as follows:

The system counter

This provides a uniform view of system time, see The system counter. Because this must be implemented at the system level, it is accessed through The system level memory-mapped implementation of the Generic Timer. However, during initialization, a status register in each implemented timer in the system must be programmed with the frequency of the system counter, so that software can read this frequency.

PE implementations of the Generic Timer

Each PE implementation of the Generic Timer provides the following components:

- A physical counter, which gives access to the count value of the system counter.
- A virtual counter, which gives access to virtual time. In AArch64 state, the CNTVOFF_EL2 register defines the offset between physical time, as defined by the value of the system counter, and virtual time.
- A number of timers. In an implementation where all Exception levels are implemented and can use AArch64 state, the timers that are accessible from AArch64 state are:
  - An EL1 physical timer.
  - A Secure EL2 physical timer.

The AArch64 view of the Generic Timer describes these components.

The system level memory-mapped implementation of the Generic Timer

The memory-mapped registers that control the components of the system level implementation of the Generic Timer are grouped into *frames*. The Generic Timer architecture defines the offset of each register within its frame, but the base address of each frame is IMPLEMENTATION DEFINED, and defined by the system.

Each system level component has one or two register frames. The possible system level components are:

The memory-mapped counter module, required

This module controls the system counter. It has two frames:

- A control frame, CNTControlBase.
- A status frame, CNTReadBase.

The memory-mapped timer control module, required

The system level implementation of the Generic Timer can provide up to eight timers, and the memory-mapped timer control module identifies:

- Which timers are implemented.
- The features of each implemented timer.

This module has a single frame, CNTCTLBase.

Memory-mapped timers, optional

An implemented memory-mapped timer:

- Must provide a privileged view of the timer, in the CNTBase*N* frame.
- Optionally. provides an unprivileged view of the timer in the CNTEL0Base*N* frame.

N is the timer number, and the corresponding frame number, in the range 0 - 7.

System Level Implementation of the Generic Timer describes these components.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D7.1.2 The system counter

The Generic Timer provides a system counter with the following specification:

Width

From Armv8.0 to Armv8.5 inclusive, at least 56 bits wide. The value returned by any 64-bit read of the counter is zero-extended to 64 bits.

Effective frequency

This indicates the amount of time each unit of the counter represents. For example:

- An effective frequency of 50MHz means each unit of the counter represents 20ns.
- An effective frequency of 1GHz means each unit of the counter represents 1ns.
- The effective frequency is indicated in the CNTFRQ_EL0 register.

From Armv8.0 to Armv8.5 inclusive, the effective frequency is a fixed value, typically in the range 1-50MHz.

Roll-over

Roll-over time of not less than 40 years.

Counter resolution

The counter resolution is a representation of how frequently the counter is updated.

For example, a counter might have an effective frequency of 1GHz, but the actual clock runs at 125MHz and therefore the counter resolution is 125Mhz.

The counter resolution might vary over time, to accommodate different operating modes of the system such as lower power modes. Such variation might be invisible to the programmer and be system specific, or might be controllable by the programmer through the frequency modes table.

Accuracy

Arm does not specify a required accuracy, but recommends that the counter does not gain or lose more than ten seconds in a 24-hour period.

Use of different counter resolutions must not affect the implemented accuracy.

Start-up

Starts operating from zero.

The system counter, once configured and running, must provide a uniform view of system time. More precisely, it must be impossible for the following sequence of events to show system time going backwards:

1. Device A reads the time from the system counter.
2. Device A communicates with another agent in the system, Device B.
3. After recognizing the communication from Device A, Device B reads the time from the system counter.

The system counter must be implemented in an always-on power domain.

To support lower-power operating modes in architectures from Armv8.0 to Armv8.5, the counter can operate with a lower counter resolution by incrementing by larger amounts at a lower actual clock frequency. For example, a system counter with an effective frequency of 10MHz might either increment:

- By 1 at 10MHz.
- By 500 at 20kHz, when the system lowers the actual clock frequency, to reduce power consumption.

In this case, the counter must support transitions between higher resolution operation and lower resolution operation, without any impact on the required accuracy of the counter.

The CNTFRQ_EL0 register is intended to hold a copy of the effective frequency to allow fast reference to this frequency by software running on the PE. For more information, see Initializing and reading the system effective frequency.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The mechanism by which the count from the system counter is distributed to system components is IMPLEMENTATION DEFINED, but each PE with a System register interface to the system counter must have a counter input that can capture each increment of the counter.

Note

So that the system counter can be clocked independently from the PE hardware, the count value might be distributed using a Gray code sequence. Gray count scheme for timer distribution scheme gives more information about this possibility.

D7.1.2.1 Initializing and reading the system effective frequency

The CNTFRQ_EL0 register must be programmed to the effective frequency of the system counter. Typically, this is done only during the system boot process, by using the System register interface to write the effective frequency to the CNTFRQ_EL0 register. Only software executing at the highest implemented Exception level can write to CNTFRQ_EL0.

Note

The CNTFRQ_EL0 register is UNKNOWN at reset, and therefore the effective frequency must be set as part of the system boot process.

Software can read the CNTFRQ_EL0 register, to determine the effective frequency, from EL2.

D7.1.2.2 Memory-mapped controls of the system counter

Some system counter controls are accessible only through the memory-mapped interface to the system counter. These controls are:

- Enabling and disabling the counter.
- Setting the counter value.
- Changing the operating mode, to change the counter resolution.
- Enabling Halt-on-debug, which a debugger can then use to suspend counting.

For descriptions of these controls, see System Level Implementation of the Generic Timer.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D7.2 The AArch64 view of the Generic Timer

The following sections describe the components and features of a PE implementation of the Generic Timer, as seen from AArch64 state:

- The physical counter.
- The virtual counter.
- Event streams.
- Timers.

D7.2.1 The physical counter

The PE includes a physical counter that contains the count value of the system counter. The CNTPCT_EL0 register holds the current physical counter value. This is defined as the pseudocode function `PhysicalCountInt`().Reads of CNTPCT_EL0 register at EL2 return the current value of `PhysicalCountInt`().

For more information, see The physical offset register.

Reads of CNTPCT_EL0 can occur speculatively and out of order relative to other instructions executed on the same PE.

D7.2.2 The virtual counter

An implementation of the Generic Timer always includes a virtual counter, which indicates virtual time.

The virtual counter contains the value of the physical counter minus a 64-bit virtual offset. When executing at EL1 or EL0, the virtual offset value relates to the current virtual machine.

The CNTVOFF_EL2 register contains the virtual offset, see The virtual offset register.

The CNTVCT_EL0 register holds the current virtual counter value.

Reads of CNTVCT_EL0 can occur speculatively and out of order relative to other instructions executed on the same PE.

D7.2.2.1 The virtual offset register

The virtual counter is a counter that has a *virtual offset* relative to the physical counter as viewed from EL2. This virtual offset is held in the register CNTVOFF_EL2. The virtual counter value is the count compared by the EL1 virtual timer.

If EL2 is not implemented, then the virtual counter uses a fixed offset of zero.

D7.2.3 Event streams

An implementation that includes the Generic Timer can use the system counter to generate one or more *event streams*, to generate periodic wakeup events as part of the mechanism described in Wait for Event.

Note

An event stream might be used:

- To impose a time-out on a Wait For Event polling loop.
- To safeguard against any programming error that means an expected event is not generated.

The CNTKCTL_EL1.{EVNTEN, EVNTDIR, EVNTI} fields define an event stream that is generated from the virtual counter.

In all implementations, the CNTHCTL_EL2.{EVNTEN, EVNTDIR, EVNTI} fields define an event stream that is generated from the physical counter.

The event stream is configured as follows:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- EVNTI selects the counter bit that triggers the event.
- EVNTI selects between bits[0:15].
- EVNTDIR selects whether the event is generated on each 0 to 1 transition, or each 1 to 0 transition, of the selected counter bit.

Note

If the event stream is configured to produce events from the low order bits of the counter when the effective frequency is very high (for example 1GHz), then the practical update rate of the counter might mean that the event stream is not generated as the low order bit might not change. Software can rely on an event stream rate of at least 1MHz in normal operation.

The pseudocode descriptions of the operation of an event stream are `SetEventRegister`, `TestEventCNTV`, and `TestEventCNTP`.

D7.2.4 Timers

In an implementation of the Generic Timer the following timers are implemented:

- An EL1 physical timer, which can be accessed from EL1.
  When this timer can be accessed from EL1, an EL1 control determines whether it can be accessed from EL0.
- As FEAT_SEL2 is always implemented, a Secure EL2 physical timer.

The output of each implemented timer:

- Provides an output signal to the system.
- If the PE interfaces to a *Generic Interrupt Controller* (GIC), signals a *Private Peripheral Interrupt* (PPI) to that GIC. In a multiprocessor implementation, each PE must use the same interrupt number for each timer.

Each timer:

- Is based around a 64-bit CompareValue that provides a 64-bit unsigned upcounter.
- Provides an alternative view of the CompareValue, called the TimerValue, that appears to operate as a 32-bit downcounter.
- Has, in addition, a 32-bit Control register.

Table D7-1 Physical timer registers summary for the Generic Timer

Timer[a] register	EL1 physical timer	Secure EL2 physical timer
CV	CNTP_CVAL_EL0	CNTHPS_CVAL_EL2
TV	CNTP_TVAL_EL0	CNTHPS_TVAL_EL2
Control	CNTP_CTL_EL0	CNTHPS_CTL_EL2

a. In this column, CV indicates the CompareValue register, and TV indicates the TimerValue register.

Table D7-2 Virtual timer register summary for the Generic Timer

Timer[a] register	EL1 virtual timer
CV	CNTV_CVAL_EL0
TV	CNTV_TVAL_EL0
Control	CNTV_CTL_EL0

a. In this column, CV indicates the CompareValue register, and TV indicates the TimerValue register.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D7.2.4.1 Operation of the CompareValue views of the timers

The CompareValue view of a timer operates as a 64-bit upcounter. The timer condition is met when the appropriate counter reaches the value programmed into its CompareValue register.

When the timer condition is met, an interrupt is generated if the interrupt is not masked in the corresponding timer control register, CNTP_CTL_EL0, or CNTV_CTL_EL0.

The operation of this view of a timer is:

```
TimerConditionMet = (((PhysicalCountInt() - Offset)[63:0] - CompareValue[63:0]) >= 0)
```

Where:

`TimerConditionMet` Is TRUE if the timer condition for this counter is met, and FALSE otherwise.

`PhysicalCountInt()` The physical counter value, which can be read from the CNTPCT_EL0 register when read from EL2.

`Offset` For the EL1 and EL2 physical timer, offset value of the EL1 physical timer is zero.

`CompareValue` The value of the appropriate CompareValue register, CNTP_CVAL_EL0, CNTHPS_CVAL_EL2 or CNTV_CVAL_EL0.

In this view of a timer, `PhysicalCountInt`(), `Offset`, and `CompareValue` are all 64-bit unsigned values.

Note

This means that a timer with a `CompareValue` of, or close to, `0xFFFF_FFFF_FFFF_FFFF` might never meet its timer condition. However, there is no practical requirement to use values close to the counter wrap value.
Software can observe the counter value by the offset in some situations by reading CNTVCT_EL0.

D7.2.4.2 Operation of the TimerValue views of the timers

The TimerValue view of a timer appears to operate as a signed 32-bit downcounter. A TimerValue register is programmed with a count value. This value decrements on each increment of the appropriate counter, and the timer condition is met when the value reaches zero. When the timer condition is met, an interrupt is generated if the interrupt is not masked in the corresponding timer control register, CNTP_CTL_EL0, CNTHP_CTL_EL2, CNTHPS_CTL_EL2, CNTPS_CTL_EL1, or CNTV_CTL_EL0.

This view of a timer depends on the following behavior of accesses to TimerValue registers:

Reads

```
TimerValue = (CompareValue[63:0] - (PhysicalCountInt()[63:0] - Offset[63:0]))[31:0]
```

Writes

```
CompareValue = ((PhysicalCountInt()[63:0] - Offset[63:0])[63:0] +
    ↪SignExtend(TimerValue[31:0]))[63:0]
```

Where the arguments other than `TimerValue` have the definitions used in Operation of the CompareValue views of the timers, and in addition:

`TimerValue` The value of a TimerValue register, CNTP_TVAL_EL0, CNTHP_TVAL_EL2, CNTHPS_TVAL_EL2, CNTPS_TVAL_EL1, or CNTV_TVAL_EL0.

In this view of a timer, values are signed in standard two's complement form.

A read of a TimerValue register after the timer condition has been met indicates the time since the timer condition was met.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

- Operation of the CompareValue views of the timers gives a strict definition of `TimerConditionMet`. However, provided that the TimerValue is not expected to wrap as a 32-bit signed value when decremented from `0x8000_0000`, the TimerValue view can be used as giving an effect equivalent to:

  `TimerConditionMet = (TimerValue ≤ 0)`

- Programming TimerValue to a negative number with magnitude greater than (Counter - Offset) can lead to an arithmetic overflow that causes the CompareValue to be a large positive value. This potentially delays meeting the timer condition for a long period of time.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter D8
The Performance Monitors Extension

This chapter describes the implementation of the Arm Performance Monitors, which are an optional non-invasive debug component. It describes version 3 of the *Performance Monitor Unit* (PMU) architecture, FEAT_PMUv3. It contains the following sections:

- About the Performance Monitors.
- Accuracy of the Performance Monitors.
- Behavior on overflow.
- Attributability.
- Controlling the PMU counters.
- Multithreaded implementations.
- Event filtering.
- Performance Monitors and Debug state.
- Counter access.
- Performance Monitors Extension registers.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.1 About the Performance Monitors

The Performance Monitors Extension is an OPTIONAL feature of an implementation, but Arm strongly recommends that implementations include version 3 of the Performance Monitors Extension, FEAT_PMUv3.

Note

No previous versions of the Performance Monitors Extension can be implemented in architectures from Armv8.0.

The Performance Monitors Extension provides:

- The 64-bit cycle counter, PMCCNTR_EL0. The cycle counter is always implemented.
- The 64-bit instruction counter, PMICNTR_EL0. The instruction counter is an optional feature, FEAT_PMUv3_ICNTR.
- The event counters, PMEVCNTR<n>_EL0. Between 0 and 31 event counters are implemented.
  The event counters are either 64 or 32 bit counters:
  - If FEAT_PMUv3p5 is implemented and the highest Exception level is using AArch64, then the event counters are 64 bit.
  - If FEAT_PMUv3p5 is not implemented, the event counters are 32 bit.
- EL2 has controls to partition the implemented event counters into the following ranges:
  - A first range which is available for use by the guest operating system accessible at all Exception levels.
  - A second range which is available for use by the hypervisor accessible EL2.

  For more information see Interaction with EL2.
- Controls for:
  - Enabling counters. See Enabling PMU counters.
  - Resetting counters. Resetting counters.
  - Flagging overflows. See Behavior on overflow.
  - Enabling interrupts on overflow. See Generating overflow interrupt requests.
  - Freezing counters. See Freezing PMU counters.
  - Prohibiting counting. See Prohibiting counting.

The PMU architecture uses event numbers to identify an event. It:

- Defines event numbers for common events, for use across many architectures and microarchitectures. See Common event numbers.

  Note

  Implementations that include FEAT_PMUv3 must, as a minimum requirement, implement a subset of the common events. See Common event numbers.
- Reserves a large event number space for IMPLEMENTATION DEFINED events. See IMPLEMENTATION DEFINED event numbers.

The full set of events for an implementation is IMPLEMENTATION DEFINED. Arm recommends that implementations include all of the events that are appropriate to the architecture profile and microarchitecture of the implementation. For more information, see PMU Event Descriptions.

When an implementation includes the Performance Monitors Extension, the architecture defines the following possible interfaces to the Performance Monitors Extension registers:

- A System register interface. This interface is mandatory.
- An external debug interface which optionally supports memory-mapped accesses. Implementation of this interface is OPTIONAL. See Recommended External Interface to the Performance Monitors.

An operating system can use the System registers to access the counters.

Also, if required, the operating system can enable application software to access the counters. This enables an application to monitor its own performance with fine-grain control without requiring operating system support. For example, an application might implement per-function performance monitoring.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

To enable interaction with external monitoring, an implementation might consider additional enhancements, such as providing:

- A set of events, from which a selection can be exported onto a bus for use as external events.
- The ability to count external events. This enhancement requires the implementation to include a set of external event input signals.

The Performance Monitors Cycle Count Register is accessible as PMCCNTR_EL0.

D8.1.1 Interaction with EL2

Software executing at or EL2 can program MDCR_EL2, Monitor Debug Configuration Register (EL2).HPMN to partition the event counters into two ranges:

- If MDCR_EL2, Monitor Debug Configuration Register (EL2).HPMN is not 0 and is less-than PMCR_EL0, Performance Monitors Control Register.N, MDCR_EL2, Monitor Debug Configuration Register (EL2).HPMN divides the event counters into a first range [0..(MDCR_EL2, Monitor Debug Configuration Register (EL2).HPMN-1)], and a second range [MDCR_EL2, Monitor Debug Configuration Register (EL2).HPMN..(PMCR_EL0, Performance Monitors Control Register.N-1)].
- If FEAT_HPMN0 is implemented and MDCR_EL2, Monitor Debug Configuration Register (EL2).HPMN is 0, all event counters are in the second range and none are in the first range.
- If MDCR_EL2, Monitor Debug Configuration Register (EL2).HPMN is equal to PMCR_EL0, Performance Monitors Control Register.N, all event counters are in the first range and none are in the second range.

Each range of event counters has its own global controls.

Counters in the second range are also described as *reserved by EL2*.

Software executing at EL2 can:

- Trap an access at EL0 or EL1 to the PMU. This means the hypervisor can identify which Guest OSs are using the PMU and intelligently employ switching of the PMU state. There is a separate trap for the PMCR_EL0, Performance Monitors Control Register register.
- If FEAT_PMUv3p1 is implemented, prohibit counting of events Attributable to EL2 by the event counters in the first range.

When EL2 is implemented and enabled in the current Security state, software executing at EL1 and, if enabled by PMUSERENR_EL0, Performance Monitors User Enable Register, EL0:

- Will read the value of MDCR_EL2, Monitor Debug Configuration Register (EL2).HPMN for PMCR_EL0, Performance Monitors Control Register.N.
- Cannot access the event counters in the second range, or the controls associated with them.

For more information, see:

- Enabling PMU counters.
- Counter access.
- Controlling the PMU counters.
- Multithreaded implementations.

D8.1.2 Interaction with trace

When FEAT_ETE is implemented, the trace unit uses the PMU events as External Inputs. See External inputs.

When FEAT_ETMv4 is implemented, it is IMPLEMENTATION DEFINED whether the implementation exports PMU events to the ETM trace unit, and the form of any exporting is also IMPLEMENTATION DEFINED. If implemented, this exporting is controlled by PMCR.X.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For more information about when the trace unit is permitted to observe PMU events, see Controls to prohibit trace at Exception levels and External inputs.

When FEAT_ETE is implemented, the PMU implements a set of events for importing external events from the trace unit and the cross-trigger interface. See Required events. Arm recommends similar functionality is implemented when FEAT_ETMv4 is implemented, and further recommends reusing the events defined for FEAT_ETE. When implemented, this functionality enables the trace unit to pass in events to be counted.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.2 Accuracy of the Performance Monitors

The Performance Monitors:

- Are a non-invasive debug component. See Non-invasive behavior.
- Must provide broadly accurate and statistically useful count information.

However, the Performance Monitors allow for:

- A reasonable degree of inaccuracy in the counts to keep the implementation and validation cost low. See A reasonable degree of inaccuracy.
- IMPLEMENTATION DEFINED controls, such as those in ACTLR registers, that software must configure before using certain PMU events. For example, to configure how the PE generates PMU events for components such as external caches and external memory.
- Other IMPLEMENTATION DEFINED controls, such as those in ACTLR registers, to optionally put the PE in an operating state that might do one or both of the following:
  - Change the level of non-invasiveness of the Performance Monitors so that enabling an event counter can impact the performance or behavior of the PE.
  - Allow inaccurate counts. This includes, but is not limited to, cycle counts.

D8.2.1 Non-invasive behavior

The Performance Monitors are a non-invasive debug feature. A non-invasive debug feature permits the observation of data and program flow. Performance Monitors, PC Sample-based Profiling and Trace are non-invasive debug features.

Non-invasive debug components do not guarantee that they do not make any changes to the behavior or performance of the processor. Any changes that do occur must not be severe however, as this will reduce the usefulness of event counters for performance measurement and profiling. This does not include any change to program behavior that results from the same program being instrumented to use the Performance Monitors, or from some other performance monitoring process being run concurrently with the process being profiled in a multitasking operating system. As such, a reasonable variation in performance is permissible.

Note

Power consumption is one measure of performance. Therefore, a reasonable variation in power consumption is permissible.

Arm does not define a *reasonable variation in performance*, but recommends that such a variation is kept within 5% of normal operating performance, when averaged across a suite of code that is representative of the application workload.

For some common architectural events, this requirement to be non-invasive can conflict with the requirement to present an accurate value of the count under normal operating conditions. Should an implementation require more performance-invasive techniques to accurately count an event, there are the following options:

- If the event is optional, define an alternative IMPLEMENTATION DEFINED event that accurately counts the event and document the impact on performance of enabling the event.
- Provide an IMPLEMENTATION DEFINED control that disables accurate counting of the event to restore broadly accurate performance, and document the impact on performance of accurate counting.

Note

The IMPLEMENTATION DEFINED invasiveness might include the PE disabling one or more performance optimizations such as multi-issue, out-of-order execution, and pipelining.
A software implication is that the performance invasiveness is proportional to the sampling interval. This can disproportionately affect profiling of rare events, because the population is smaller and therefore the interval will tend to be smaller.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.2.2 A reasonable degree of inaccuracy

The Performance Monitors provide broadly accurate and statistically useful count information. To keep the implementation, validation, and performance overhead costs low, a reasonable degree of inaccuracy in the counts and the detection of overflow is acceptable. Arm does not define a *reasonable degree of inaccuracy* but recommends the following guidelines:

- Under normal operating conditions the counters must present an accurate value of the count.
- In exceptional circumstances, such as enabling or disabling the counter, a change in Security state, or other boundary condition, it is acceptable for the count to be inaccurate.
- In exceptional microarchitectural circumstances, including when the event occurs rapidly in close succession, the overflow of a counter in synchronous mode might only be observed by a later instruction that also generates the event.

  This means that there might be temporal inaccuracy in the generation of the PMU exception. This is also known as the shadow effect.
- Under very unusual, non-repeating pathological cases, the counts can be inaccurate. These cases are likely to occur as a result of asynchronous exceptions, such as interrupts, where the chance of a systematic error in the count is very unlikely.

Note

An implementation must not introduce inaccuracies that can be triggered systematically by the execution of normal pieces of software. For example, it is not reasonable for the count of branch behavior to be inaccurate when caused by a systematic error generated by the loop structure producing a dropping in branch count.
However, dropping a single branch count as the result of a rare interaction with an interrupt is acceptable.

The architecture does not permit these behaviors for events that specifically count architectural exceptional circumstances, such as a change in Exception level, for example, EXC_TAKEN.

The permitted inaccuracy limits the possible uses of the Performance Monitors. In particular, the architecture does not define the point in a pipeline where the event counter is incremented, relative to the point where a read of the event counters is made. This means that pipelining effects can cause some imprecision, and can affect which events are counted.

This applies to both architectural and microarchitectural events, and can apply differently to different events. Arm recommends that is applied consistently between counters. For example, any boundary condition effects that apply to an event counter counting the CPU_CYCLES event should apply in the same way to the cycle counter PMCCNTR_EL0. However, this is not required.

Inaccuracies around exceptional events, such as enabling or disabling the counter, should be such that the inaccuracy is reduced by lengthening the sampling period.

Where a direct write to a Performance Monitors control register disables a counter, and is followed by a Context Synchronization event, any subsequent indirect read of the control register by the Performance Monitors to determine whether the counter is enabled will return the updated value. Any subsequent direct read of the counter or counter overflow status flags will return the value at the point the counter was disabled.

Note

The imprecision means that the counter might have counted an event around the time the counter was disabled, but does not allow the event to be observed as counted after the counter was disabled.

In addition to this, entry to and exit from Debug state can disturb the normal running of the PE, causing further inaccuracy in the Performance Monitors. Disabling the counters while in Debug state limits the extent of this inaccuracy. An implementation can employ methods to limit this inaccuracy, for example by promptly disabling the counters during the Debug state entry sequence.

An implementation must document any particular scenarios where significant inaccuracies are expected.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.3 Behavior on overflow

The event counters, PMEVCNTR<n>_EL0, Performance Monitors Event Count Registers, n = 0 - 30 are 32-bit unsigned counters that overflow if incrementing PMEVCNTR<n>_EL0, Performance Monitors Event Count Registers, n = 0 - 30 causes an unsigned overflow of an event counter, the PE sets PMOVSCLR_EL0, Performance Monitors Overflow Flag Status Clear Register[*n*] to 1.

The cycle counter, PMCCNTR_EL0, Performance Monitors Cycle Count Register, is a 64-bit unsigned counter, f incrementing PMCCNTR_EL0, Performance Monitors Cycle Count Register causes an unsigned overflow of bits [31:0] of the cycle counter, the PE sets PMOVSCLR_EL0, Performance Monitors Overflow Flag Status Clear Register[31] to 1.

When any overflow occurs, an interrupt request is generated if the PE is configured to generate counter overflow interrupts. For more information, see Generating overflow interrupt requests.

The update of PMOVSCLR_EL0, Performance Monitors Overflow Flag Status Clear Register occurs synchronously with the update of the counter.

A PMU overflow can be used as an input trigger event to the *Cross Trigger Interface* (CTI) for the PE. For more information, see Performance Monitors overflow trigger event.

D8.3.1 Generating overflow interrupt requests

Software can program the Performance Monitors so that an overflow interrupt request is generated when a counter overflows. See PMINTENSET_EL1, Performance Monitors Interrupt Enable Set Register and PMINTENCLR_EL1, Performance Monitors Interrupt Enable Clear Register.

A PMU interrupt request is always taken asynchronously.

Note

- The mechanism by which an interrupt request from the Performance Monitors generates an FIQ or IRQ exception is IMPLEMENTATION DEFINED.
- Arm recommends that the overflow interrupt requests:
  - Translate into a **PMUIRQ** signal, so that they are observable to external devices.
  - Connect to inputs on an IMPLEMENTATION DEFINED Generic Interrupt Controller as a *Private Peripheral Interrupt* (PPI) for the originating processor. See the *ARM Generic Interrupt Controller Architecture Specification* for information about PPIs.
  - Connect to a CTI, see The Embedded Cross-Trigger Interface.
- Arm strongly discourages implementations from connecting overflow interrupt requests from multiple PEs to the same *System Peripheral Interrupt* (SPI) identifier.
- From GICv3, the *ARM® Generic Interrupt Controller Architecture Specification* recommends that the *Private Peripheral Interrupt* (PPI) with ID 23 is used for overflow interrupt requests.

Software can write to the counters to control the frequency at which interrupt requests occur. For example, software might set a 32-bit counter to `0xFFFF_0000`, to generate another counter overflow after 65 536 increments, and reset it to this value every time an overflow interrupt occurs.

Note

If an event can occur multiple times in a single clock cycle, then counter overflow can occur without the counter registering a value of zero.

The overflow interrupt request is a level-sensitive request. The PE signals a request for:

- Any implemented PMEVCNTR<n>_EL0, Performance Monitors Event Count Registers, n = 0 - 30 counter, when the global enable control for the event counter is 1, the value of PMOVSSET_EL0, Performance Monitors Overflow Flag Status Set Register[*n*] is 1, and the value of PMINTENSET_EL1, Performance Monitors Interrupt Enable Set Register[*n*] is 1.
- The cycle counter, when the values of PMCR_EL0.E, PMOVSSET_EL0, Performance Monitors Overflow Flag

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Status Set Register[31] and PMINTENSET_EL1, Performance Monitors Interrupt Enable Set Register[31] are all 1.

- The instruction counter, when the values of PMCR_EL0.E, PMOVSSET_EL0.F0, and PMINTENSET_EL1.F0, are all 1.

The overflow interrupt request is active in all Security states.

The interrupt handler for the counter overflow request must cancel the interrupt request, by writing 1 to PMOVSCLR_EL0, Performance Monitors Overflow Flag Status Clear Register[*n*] to clear the overflow bit to 0.

D8.3.1.1 Pseudocode description of overflow interrupt requests

See R-profile Architecture Pseudocode for a pseudocode description of overflow interrupt requests. The `CheckForPMUOverflow()` pseudocode function signal PMU overflow interrupt requests to an interrupt controller and PMU overflow trigger events to the cross-trigger interface.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.4 Attributability

An event caused by the PE counting the event is *Attributable*. If an agent other than the PE that is counting the events causes an event, these events are Unattributable.

An event is defined as being either Attributable or Unattributable. If the event is Attributable, it is further defined whether it is Attributable to:

- The current Security state of the PE.
- The current Exception level of the PE.
- When the PE is in Debug state, operations issued to the PE by the debugger through the external debug interface.

In a multithreaded implementation, an event might be generated by another PE with the same values for affinity level 1 and higher. This event is further defined as Attributable to:

- The current Security state of that PE.
- The current Exception level of that PE.
- When that PE is in Debug state, operations issued to that PE by the debugger through the external debug interface.

See Multithreaded implementations for information about enabling and restricting counting events in a multithreaded implementation.

Note

An implementation is described as multithreaded when the lowest level of affinity, as reported by MPIDR_EL1{Aff3, Aff2, Aff1, Aff0}, consists of logical PEs that are implemented using a multithreading type approach. In this section, when referring to a multithreaded implementation, *thread* is used to mean processing elements with:

- MPIDR_EL1.MT set to 1,
- Different values for affinity level 0.
- The same values for affinity level 1 and higher.

An event can be defined as the combination of multiple subevents, which can be either Attributable or Unattributable.

All architecturally defined events are Attributable, unless otherwise stated.

Unattributable events might be counted when Attributable events are not counted. See:

- Event filtering.
- Performance Monitors and Debug state.
- Counting events from shared components.

These sections are summarized by Table D8-1 for events Attributable to the processor, and Unattributable events. Table D8-1 entries apply when the counter and PMU are enabled and not frozen. Otherwise, events are not counted.

Table D8-1 Counting events

State	Allowed or prohibited	Filtered	Event type		
			If Attributable to:	Then	Else
Non- debug	Allowed	Not filtered	X	Count	Count
		Filtered	Current Exception level	Do not count	IMPLEMENTATION DEFINED
	Prohibited	X	Current Security state	Do not count	IMPLEMENTATION DEFINED
Debug	X	X	Debugger operations or raw cycles	Do not count	IMPLEMENTATION DEFINED

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.5 Controlling the PMU counters

This section describes the mechanisms available for controlling the PMU event and cycle counters. The following sections describe those mechanisms:

- Enabling PMU counters.
- Freezing PMU counters.
- Prohibiting counting.

D8.5.1 Enabling PMU counters

Counters are enabled by a combination of the *global enable control* for the counter and a counter enable control, PMCNTENSET_EL0, Performance Monitors Count Enable Set Register, as described in the following sections.

D8.5.1.1 Enabling the event counters

PMCNTENSET_EL0, Performance Monitors Count Enable Set Register.P<n> is the counter enable control for event counter *n*.

The counters are controlled by the following:

- For an event counter *n* reserved by EL2, the global enable control is MDCR_EL2, Monitor Debug Configuration Register (EL2).HPME.
- For all other event counters, the global enable control is PMCR_EL0, Performance Monitors Control Register.E.

For more information, see Interaction with EL2.

Table D8-2 shows the combined effect of all the counter enable controls.

Table D8-2 Event counter enables when an implementation includes EL2

MDCR_EL2.HPME	PMCR_EL0.E	PMCNTENSET_EL0.P[*n*] == 0	PMCNTENSET_EL0.P[*n*] == 1	
			n < MDCR_EL2.HPMN	*n* ≥ MDCR_EL2.HPMN
0	0	PMEVCNTR<n>_EL0 disabled	PMEVCNTR<n>_EL0 disabled	PMEVCNTR<n>_EL0 disabled
0	1	PMEVCNTR<n>_EL0 disabled	PMEVCNTR<n>_EL0 enabled	PMEVCNTR<n>_EL0 disabled
1	0	PMEVCNTR<n>_EL0 disabled	PMEVCNTR<n>_EL0 disabled	PMEVCNTR<n>_EL0 enabled
1	1	PMEVCNTR<n>_EL0 disabled	PMEVCNTR<n>_EL0 enabled	PMEVCNTR<n>_EL0 enabled

D8.5.1.2 Enabling the cycle counter

For the cycle counter, PMCCNTR_EL0, Performance Monitors Cycle Count Register, the global enable control is PMCR_EL0, Performance Monitors Control Register.E.

For the cycle counter, the counter enable control is PMCNTENSET_EL0, Performance Monitors Count Enable Set Register.C.

PMCR_EL0, Performance Monitors Control Register.DP controls whether the cycle counter is disabled when event counting is prohibited, as described in Prohibiting counting.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.5.1.3 Enabling the instruction counter

When FEAT_PMUv3_ICNTR is implemented, for the instruction counter, PMICNTR_EL0, the global enable control is PMCR_EL0, Performance Monitors Control Register.E and the counter enable control is PMCNTENSET_EL0.F0.

D8.5.2 Freezing PMU counters

The cycle counter, PMCCNTR_EL0, is disabled when an unsigned overflow occurs, PMCR_EL0.FZO is 1, and PMCR_EL0.DP is 1.

Note

The architecture does not define when PMU events are counted relative to the instructions that caused the event. Events caused by an instruction might be counted before or after the instruction becomes architecturally executed, and events might be counted for operations that are not architecturally executed. Events can be counted speculatively, out-of-order, or both with respect to the simple sequential execution of the program. Events might also be counted simultaneously by other event counters when the overflow occurs, including events from different instructions. Multiple instances of an event might occur simultaneously, thus an event counter unsigned overflow can yield a nonzero value in the event counter. Arm recommends that such counting anomalies are minimized when software uses the freeze on overflow feature. When the freeze on overflow feature is being used, software cannot assume that the event counter stops counting at zero when an overflow occurs.

To freeze the PMU counter values on taking a synchronous PMU exception, software should configure the relevant PMU counters to not count at the Exception level where the PMU exception is taken to and higher Exception levels. This has the effect of stopping counting when the PMU exception is taken.

Note

This mechanism can only be used when profiling software that only executes at lower Exception levels.

If an event counter *n* overflows, where *n* is even and event counter *n*+1 is configured to count the CHAIN event, it is CONSTRAINED UNPREDICTABLE whether the CHAIN event observes the overflow event when the applicable PMCR_EL0.FZO or MDCR_EL2.HPMFZO bit is 1 and the corresponding PMCR_EL0.LP or MDCR_EL2.HLP bit is 0.

If a direct read of PMOVSCLR_EL0 returns a nonzero value for a subset of the overflow flags, which means an event counter *n* should not count, then a sequence of direct reads of PMEVCNTR<n>_EL0 ordered after the read of PMOVSCLR_EL0 and before the PMOVSCLR_EL0 flags are cleared to zero, will return the same value for each read, because the event counter has stopped counting.

Note

Direct reads of System registers require explicit synchronization for following direct reads of other System registers to be ordered after the first direct read.

D8.5.3 Resetting counters

Software can reset all the event counters and/or the cycle counter in a single operation by writing 1 to PMCR_EL0, Performance Monitors Control Register.C and/or PMCR_EL0, Performance Monitors Control Register.P.

If FEAT_PMUv3p9 is implemented, then counters can be individually reset together in a single operation using PMZR_EL0.

D8.5.4 Prohibiting counting

Counting Attributable events is prohibited at EL2 unless any of the following is true:

- FEAT_PMUv3p1 is not implemented.
- MDCR_EL2, Monitor Debug Configuration Register (EL2).HPMD is 0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- MDCR_EL2.HPMN is less than PMCR_EL0.N, the event is being counted by event counter *n*, and *n* is in the range [MDCR_EL2.HPMN .. (PMCR_EL0.N-1)].

If FEAT_SEL2 is implemented, counting Attributable events at Secure EL2 is allowed if and only if counting events is allowed in Secure state, and counting events is allowed at EL2.

For each Unattributable event, it is IMPLEMENTATION DEFINED whether it is counted when counting Attributable events is prohibited.

The accessibility of Performance Monitors registers is unaffected by whether event counting is enabled or prohibited.

All of the following apply to the instruction counter, PMICNTR_EL0:

- Counting is prohibited at EL2 unless MDCR_EL2.HPMD is 0.

The cycle counter, PMCCNTR_EL0, Performance Monitors Cycle Count Register, counts unless any of the following is true:

- The cycle counter is disabled by PMCR_EL0.E or PMCNTENSET_EL0.C.
- PMCR_EL0, Performance Monitors Control Register.DP is 1 and one of the following is true:
  - Event counting by event counters in the range [0 .. (MDCR_EL2, Monitor Debug Configuration Register (EL2).HPMN-1)] is prohibited.
  - Event counting by event counters in the range [0 .. (MDCR_EL2, Monitor Debug Configuration Register (EL2).HPMN-1)] is frozen by PMCR_EL0, Performance Monitors Control Register.FZO.
- The PE is in Debug state.

See `CountPMUEvents()` in R-profile Architecture Pseudocode for more information. The `CountPMUEvents(n)` function returns TRUE if PMEVCNTR<n>_EL0, Performance Monitors Event Count Registers, n = 0 - 30 is enabled and allowed to count events at the current Exception level or state, and FALSE otherwise. The function `CountPMUEvents(31)` returns TRUE if the cycle counter is enabled and allowed to count cycles at the current Exception level and state, and FALSE otherwise. The function `CountPMUEvents(32)` returns TRUE if the instruction counter is enabled and allowed to count instructions at the current Exception level and state, and FALSE otherwise. However, these functions do not completely describe the behavior for Unattributable events.

The Performance Monitors are intended to be broadly accurate and statistically useful, see Accuracy of the Performance Monitors. Some inaccuracy is permitted at the point of changing between a state where counting is prohibited and a state where counting is allowed, however. To avoid the leaking of information, the permitted inaccuracy is that transactions that are not prohibited can be uncounted. Where possible, prohibited transactions must not be counted, but if they are counted, then that counting must not degrade security.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.6 Multithreaded implementations

If an implementation is multithreaded and the Effective value of PMEVTYPER<n>.MT is 1, then, unless otherwise stated, the counter counts the sum of all events across all PEs with the same level 1 Affinity. A pair of PEs have the same level 1 Affinity if they have the same values for all fields in MPIDR_EL1or MPIDR except the Aff0 field.

See also Cycle event counting in multithreaded implementations for additional requirements for events that count cycles.

Events on other PEs are not counted when the Effective value of PMEVTYPER<n>_EL0, Performance Monitors Event Type Registers, n = 0 - 30.MT is 0.

If the CPU implements multithreading, it is IMPLEMENTATION DEFINED whether PMEVTYPER<n>_EL0, Performance Monitors Event Type Registers, n = 0 - 30.MT is implemented as RW or RES0.

When the current configuration is not multithreaded, and PEA prohibits counting of events Attributable to Secure state when PEA is in Secure state, it is IMPLEMENTATION DEFINED whether:

- Counting events Attributable to Secure state when PEA is in Non-secure state is permitted.
- Counting Unattributable events related to other Secure operations in the system when PEA is in Non-secure state is permitted.

Otherwise, counting events in Non-secure state is permitted.

When the current configuration is not multithreaded, and PEA prohibits counting of events Attributable to EL2 when PEA is at EL2, it is IMPLEMENTATION DEFINED whether:

- Counting events Attributable to EL2 when PEA is using another Exception level is permitted.
- Counting Unattributable events related to EL2 when PEA is using another Exception level is permitted.

Otherwise, counting events at another Exception level is permitted.

When the current configuration is not multithreaded, and PEA prohibits counting of events Attributable to a Security state and an Exception level when FEAT_PMUv3p7 is implemented, it is IMPLEMENTATION DEFINED whether:

- Counting events Attributable to EL2 when PEA is using another Exception level is permitted.
- Counting Unattributable events related to EL2 when PEA is using another Exception level is permitted.

See Time as measured by the Performance Monitors cycle counter.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.7 Event filtering

The PMU can filter events by various combinations of Exception level and Security state. This gives software the flexibility to count events across multiple processes.

D8.7.1 Filtering by Exception level and Security state

In AArch64 state:

- For each event counter, PMEVTYPER<n>_EL0 specifies the Exception levels in which the counter counts events Attributable to Exception levels.
- PMCCFILTR_EL0 specifies the Exception levels in which the cycle counter counts.

For an event that is Attributable to an Exception level, in a multithreaded implementation:

- When the Effective value of PMEVTYPER<n>_EL0.MT is 1, the specified filtering is evaluated using the current Exception level and Security state of the thread to which the event is Attributable. See Example D8-1.
- When the Effective value of PMEVTYPER<n>_EL0.MT is 0, the event is only counted if it is Attributable to the counting thread, and the filtering is evaluated using the Exception level and Security state of the counting thread.

Example D8-1 Example of the effect of the PMEVTYPER<*n*>_EL0.MT control

In a multithreaded implementation, if the Effective value of PMEVTYPER<n>_EL0.MT is 1 and the value of PMEVTYPER<n>_EL0.U is 1 on the counting thread, then event counter *n* does not count events Attributable to EL0 on another thread, even if the counting thread is not executing at EL0.

For each Unattributable event, it is IMPLEMENTATION DEFINED whether the filtering applies. In a multithreaded implementation, if the filtering applies to an Unattributable event, then the filtering is evaluated using the Exception level and Security state of the counting thread.

For more information, see the individual register descriptions and Multithreaded implementations.

D8.7.2 Accuracy of event filtering

For most events, it is acceptable that, during a transition between states, events generated by instructions executed in one state are counted in the other state. The following sections describe the cases where event counts must not be counted in the wrong state:

- Exception-related events.
- Software increment events.

D8.7.2.1 Exception-related events

The PMU must filter events related to exceptions and exception handling according to the Exception level in which the event occurred. These events are:

- EXC_TAKEN, Exception taken.
- EXC_RETURN, Instruction architecturally executed, Condition code check pass, exception return.
- CID_WRITE_RETIRED, Instruction architecturally executed, Condition code check pass, write to CONTEXTIDR_EL1, Context ID Register (EL1).
- TTBR_WRITE_RETIRED, Instruction architecturally executed, Condition code check pass, write to translation table base.
- EXC_UNDEF, Exception taken, other synchronous.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- EXC_SVC, Exception taken, Supervisor Call.
- EXC_PABORT, Exception taken, Instruction Abort.
- EXC_DABORT, Exception taken, Data Abort or SError.
- EXC_IRQ, Exception taken, IRQ.
- EXC_FIQ, Exception taken, FIQ.
- EXC_SMC, Exception taken, Secure Monitor Call.
- EXC_HVC, Exception taken, Hypervisor Call.
- EXC_TRAP_PABORT, Exception taken, Instruction Abort not Taken locally.
- EXC_TRAP_DABORT, Exception taken, Data Abort or SError not Taken locally.
- EXC_TRAP_OTHER, Exception taken, other traps not Taken locally.
- EXC_TRAP_IRQ, Exception taken, IRQ not Taken locally.
- BRB_FILTRATE, Branch record captured.

The PMU must not count an exception after it has been taken because this could systematically report a result of zero exceptions at EL0. Similarly, it is not acceptable for the PMU to count exception returns or writes to CONTEXTIDR_EL1, Context ID Register (EL1) after the return from the exception.

If FEAT_SEBEP is implemented, then when any precise exception is taken, including PMU exceptions, the value of all event counters counting at-retirement events and the instruction counter are precise. See R_{TNVSL} in Definition of a precise exception and imprecise exception.

D8.7.2.2 Software increment events

The PMU must filter software increment events according to the Exception level in which the software increment occurred. Software increment counting must also be precise, meaning the PMU must count every architecturally executed software increment event, and must not count any Speculatively executed software increment.

If the PE performs two architecturally executed writes to the PMSWINC_EL0 register without an intervening Context Synchronization event, then the counter is incremented twice.

For more information, see SW_INCR.

D8.7.3 Pseudocode description of event filtering

See `CountPMUEvents()` in R-profile Architecture Pseudocode for a pseudocode description of event filtering. However, this function does not completely describe the behavior for Unattributable events.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.8 Performance Monitors and Debug state

Events that count cycles are not counted in Debug state.

Events Attributable to the operations issued by the debugger through the external debug interface are not counted in Debug state.

In an implementation that supports multithreading, when the Effective value of PMEVTYPER<n>_EL0.MT is 1, if an event is Attributable to an operation issued by the debugger through the external debug interface to another thread that is in Debug state, then the event is not counted, and it is IMPLEMENTATION DEFINED whether the event is counted when the counting thread is in Debug state.

For each Unattributable event, it is IMPLEMENTATION DEFINED whether it is counted when the counting PE is in Debug state. If the event might be counted, then the rules in Filtering by Exception level and Security state apply for the current Security state in Debug state.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.9 Counter access

All implemented event counters are accessible in EL2. If EL2 is implemented the hypervisor uses MDCR_EL2, Monitor Debug Configuration Register (EL2).HPMN to reserve an event counter, with the effect that if EL2 is enabled in the current Security state, software cannot access that counter and its associated state from EL0 or EL1.

Note

Whether software can access an event counter at an Exception level does not affect whether the counter counts events at that Exception level. For more information, see Controlling the PMU counters and Enabling PMU counters.

D8.9.1 PMEVCNTR<*n*> event counters

Table D8-3 shows how the number of implemented event counters, PMCR_EL0, Performance Monitors Control Register.N, and if EL2 is implemented, the value of the MDCR_EL2, Monitor Debug Configuration Register (EL2).HPMN field affects the behavior of permitted accesses to the PMEVCNTR<n>_EL0, Performance Monitors Event Count Registers, n = 0 - 30 event counter registers for values of *n* from 0 to 30.

Table D8-3 Result of PMEVCNTR<n> event counter accesses

Condition	Access at Exception level		
	EL2	EL1	EL0
n < HDCR.HPMN and EL2 is implemented and enabled in the current Security state	Succeeds	Succeeds	Succeeds
n ≥ MDCR_EL2, Monitor Debug Configuration Register (EL2).HPMN and *n* < PMCR_EL0, Performance Monitors Control Register.N and EL2 is implemented and enabled in the current Security state	Succeeds	No access	No access
n ≥ PMCR_EL0, Performance Monitors Control Register.N	No access	No access	No access

Where Table D8-3 shows access succeeds for an event counter *n*, the access might be UNDEFINED or generate a trap exception. See the descriptions of PMEVCNTR<n>_EL0, Performance Monitors Event Count Registers, n = 0 - 30 and PMXEVCNTR_EL0, Performance Monitors Selected Event Count Register for details.

Where Table D8-3 shows no access for an event counter *n*:

- When PMSELR_EL0, Performance Monitors Event Counter Selection Register.SEL is *n*, the PE prevents direct reads and direct writes of PMXEVTYPER_EL0, Performance Monitors Selected Event Type Register or PMXEVCNTR_EL0, Performance Monitors Selected Event Count Register. See the register descriptions for more information.
- The PE prevents direct reads and direct writes of PMEVTYPER<n>_EL0, Performance Monitors Event Type Registers, n = 0 - 30 or PMEVCNTR<n>_EL0, Performance Monitors Event Count Registers, n = 0 - 30. See the register descriptions for more information.
- If FEAT_PMUv3_SS is implemented, the PE prevents direct reads and direct writes of PMEVCNTSVR<n>_EL1. See the register descriptions for more information.
- Direct reads and direct writes of the following register fields are RAZ/WI: PMOVSCLR_EL0, Performance Monitors Overflow Flag Status Clear Register[*n*], PMOVSSET_EL0, Performance Monitors Overflow Flag Status Set Register[*n*], PMCNTENSET_EL0, Performance Monitors Count Enable Set Register[*n*], PMCNTENCLR_EL0, Performance Monitors Count Enable Clear Register[*n*], PMINTENSET_EL1, Performance Monitors Interrupt Enable Set Register[*n*], PMINTENCLR_EL1, Performance Monitors Interrupt Enable Clear Register[*n*], and, when FEAT_PMUv3p9 is implemented, PMUACR_EL1[n].
- Direct writes to PMSWINC_EL0, Performance Monitors Software Increment Register[*n*] are ignored, and, when FEAT_PMUv3p9 is implemented, direct writes to PMZR_EL0[n] are ignored.
- A direct write of 1 to PMCR_EL0, Performance Monitors Control Register.P does not reset PMEVCNTR<n>_EL0, Performance Monitors Event Count Registers, n = 0 - 30.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.9.2 Cycle and instruction counters

The PMU does not provide any control that a hypervisor can use to reserve the cycle counter for its own use. However, access to the PMU registers are subject to the access permissions described in Configurable instruction controls.

PMICNTR_EL0 counts in AArch64 state.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D8.10 Performance Monitors Extension registers

Further information on the Performance Monitors Extension Registers can be found in the following sections:

- Performance monitors registers summarizes the Performance Monitors Extension registers in AArch64 state.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter D9
PMU Event Descriptions

This chapter describes the PMU events that can be counted, their associated event numbers, and the mnemonics for the events. It contains the following sections:

- Definitions.
- The PMU event number space and common events.
- Common event numbers.
- Cycle event counting.
- Meaningful combinations of common events.
- Required events.
- Synchronous events.
- IMPLEMENTATION DEFINED event numbers.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D9.1 Definitions

The following subsections give more information about terms used in the event definitions:

- Definition of terms.
- Levels of caches and TLBs.
- Counting events from shared components.
- Counting exceptions taken locally or not taken locally.

D9.1.1 Definition of terms

This section describes terms used by PMU events as they relate to the implementation of the PMU. For more definitions, see Glossary.

Architectural event

An architectural event is an event which gives the same result for the same program on any implementation of the Arm architecture, subject to the program having the same inputs, including asynchronous events triggered by the system, any IMPLEMENTATION DEFINED or UNPREDICTABLE variation permitted by the event definition, and the reasonable degree of inaccuracy described in A reasonable degree of inaccuracy. INST_RETIRED is an example of an architectural event.

An architectural event should not be confused with an event defined by the architecture, which is referred to as a Common event.

Note: Do not confuse Architectural events and Common events. An Architectural event can be either Common or IMPLEMENTATION DEFINED. See also Common event and IMPLEMENTATION DEFINED event.

At-retirement event

An at-retirement event is generated by an Architecturally executed instruction that does not generate an exception, unless specified otherwise. This includes SVC, HVC, and SMC exceptions.

For cache and TLB events:

- The event counts each architecturally-executed instruction that generates the event.
- The event is only counted once, by the first instruction that causes it. This might not be the first instruction in program order.

  Note

  Consider two loads that execute out of order that access the same cache line granule which is not in the cache. The second instruction in program order causes the miss, and the first instruction fetches data from the same cache miss result. Only the second instruction counts the cache miss event. Also, the second instruction might be speculative and the first instruction retired, meaning the cache miss is not counted at all.

- The population counts for the events are the architecturally-executed instruction counts.

Common event

An event that is defined by the Performance Monitors Extension. For more information, see Common event numbers.

Note: Do not confuse Common events and Architectural events. A Common event can be either Architectural or Microarchitectural. See also Architectural event and Microarchitectural event.

CSIZE

Container size, in bits, that corresponds to the largest non-overlapping Advanced SIMD vector element size or scalar register size that is encoded in the instruction opcode.

Event in progress

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Some events count when another event or condition is *in progress*. This might mean that the event counts the occupancy of a queue or other microarchitectural structure tracking the event. It is usually IMPLEMENTATION DEFINED when an event is in progress.

For example, the MEM_ACCESS_RD_PERCYC event counts when the MEM_ACCESS_RD event is in progress, meaning on each Processor cycle, the counter increments by the number of Memory-read operations that are in progress.

These events can be used to calculate the average number of events in progress. In the case of the MEM_ACCESS_RD_PERCYC event, this is also the average read latency. However, the definition of *in progress* might not include all parts of the operation.

Example D9-1 Stages of an operation not included in the definition of event in progress

In an example implementation, a Memory-read operation generated by a load instruction occupies three pipeline stages in the PE before generating a MEM_ACCESS_RD event when the PE starts to access memory. The event is then considered to be *in progress* until the data is returned to the PE. In the case of a Normal Cacheable access, the PE first looks in the Level 1 data cache, and if the address is cached, returns data in two cycles. Once the data is returned, it is another cycle before the result can be forwarded to any other instruction.
In this example, if all loads hit in the Level 1 cache, the average read latency calculated using the MEM_ACCESS_RD_PERCYC and MEM_ACCESS_RD events might be two cycles. Although this value is correct for the IMPLEMENTATION SPECIFIC definition of this event, it has to be adjusted by a constant four additional cycles to match the more commonly understood definition of Level 1 cache access latency, which for this example would be quoted as six cycles.

Instruction architecturally executed

An instruction that is part of the Execution stream. See also Architecturally executed.

A reasonable degree of inaccuracy allows for counts to be inaccurate in exceptional circumstances. For an event that counts instructions architecturally executed, this allows an implementation to count instructions that do not form part of the Execution stream because of an exceptional event, such as if the instruction generates a synchronous exception or entry to Debug state.

Instructions that have no visible effect on the architectural state of the PE are architecturally executed and counted even if they form part of the Execution stream.

Examples of instructions that have no visible effect are:

- A `NOP`.
- A conditional instruction that fails its Condition code check.
- A Compare and Branch on Zero, `CBZ`, instruction that does not branch.
- A Compare and Branch on Nonzero, `CBNZ`, instruction that does not branch.

However, for events that count only the execution of instructions that update System registers, such as CID_WRITE_RETIRED, if such an instruction is executed twice without an intervening Context Synchronization event, it is constrained unpredictable whether the first instruction is counted.

Instruction architecturally executed, Condition code check pass

Instruction architecturally executed, Condition code check pass is a class of events that explicitly do not occur for:

- A conditional instruction that fails its Condition code check.
- A Compare and Branch on Zero, `CBZ`, instruction that does not branch.
- A Compare and Branch on Nonzero, `CBNZ`, instruction that does not branch.
- A Test and Branch on Zero, `TBZ`, instruction that does not branch.
- A Test and Branch on Nonzero, `TBNZ`, instruction that does not branch.
- A Store-Exclusive instruction that does not write to memory.

Otherwise, the definition of architecturally executed is the same as for Instruction architecturally executed.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A branch that is architecturally executed, with condition code check pass is also described as a branch taken.

Instruction memory access

A PE acquires instructions for execution through instruction fetches. Instruction fetches might be due to:

- Fetching instructions that are architecturally executed.
- The result of the execution of an instruction prefetch instruction, `PLI`.
- Speculation that a particular instruction might be executed in the future.

The relationship between the fetch of an individual instruction and an instruction memory access is IMPLEMENTATION DEFINED. For example, an implementation might fetch many instructions including a non-integer number of instructions in a single instruction memory access.

IMPLEMENTATION DEFINED event

An event that is defined by the implementation. For more information, see IMPLEMENTATION DEFINED event numbers.

Note: Do not confuse IMPLEMENTATION DEFINED events and Microarchitectural events. An IMPLEMENTATION DEFINED event can be either Architectural or Microarchitectural. See also Architectural event and Microarchitectural event.

Memory-read operations

A PE accesses memory through memory-read operations and Memory-write operations. A memory-read operation might be due to:

- The result of an architecturally executed Memory-reading instructions.
- The result of a Speculatively executed memory-reading instructions.
- A translation table walk.

For levels of cache hierarchy beyond the Level 1 caches, memory-read operations also include accesses made as part of a refill of another cache closer to the PE. Such refills might be due to:

- Memory-read operations or Memory-write operations that miss in the cache
- The execution of a data prefetch instruction.
- The execution of a cache maintenance instruction.

  Note

  A prefetch instruction or cache maintenance instruction is not, in itself, an access to that cache. However, it might generate cache refills which are then treated as memory-read operations beyond that cache.

- Speculation that a future instruction might access the memory location.
- Instruction memory accesses to a unified cache, when the cache does not implement the L<n>I_CACHE event.

This list is not exhaustive.

The relationship between memory-read instructions and memory-read operations is IMPLEMENTATION DEFINED. For example, for some implementations an `LDP` instruction that reads two 64-bit registers might generate one memory-read operation if the address is quadword-aligned, but for other addresses it generates two or more memory-read operations.

Memory-reading instructions

An instruction with explicit Memory read effects. Loads and atomic instructions that return a value to the PE are counted as Memory-reading instructions.

Unless otherwise specified, this includes instructions that can have a Memory read effect, even if the instruction does not actually read from memory for some other reason. For example, a load that generates an MMU fault.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

It is IMPLEMENTATION DEFINED whether the prefetch instructions PRFM, PLD, PLDW, and PLI count as integer data-processing instructions or Memory-reading instructions. Arm recommends that if the instruction is not implemented as a NOP then it is counted as a Memory-reading instruction.

Memory-writing instructions, other than atomics, that return a result in a PE register other than data from the location being accessed are not counted as Memory-reading instructions. This includes Store-exclusive instructions.

Atomic instructions that do not return a value to the PE are not counted as Memory-reading instructions.

Memory-write operations

Memory-write operations might be due to:

- The result of an architecturally executed Memory-writing instructions.
- The result of a Speculatively executed Memory-writing instructions.
- A hardware update of a translation table entry.

Note

Speculatively executed Memory-writing instructions that do not become architecturally executed must not alter the architecturally defined view of memory. They can, however, generate a Memory-write operation that is later undone in some IMPLEMENTATION SPECIFIC way.

For levels of cache hierarchy beyond the Level 1 caches, Memory-write operations also include accesses made as part of a write-back from another cache closer to the PE. Such write-backs might be due to:

- Evicting a dirty line from the cache, to allocate a cache line for a cache refill, see Memory-read operations.
- The execution of a cache maintenance instruction.

  Note

  A cache maintenance instruction is not in itself an access to that cache. However, it might generate write-backs which are then treated as Memory-write operations beyond that cache.

- The result of a coherency request from another PE.

This list is not exhaustive.

DC ZVA is counted as a Memory-write operation.

The relationship between Memory-writing instructions and Memory-write operations is IMPLEMENTATION DEFINED. For example, for some implementations an STP instruction that writes two 64-bit registers might generate one Memory-write operation if the address is quadword-aligned, but for other addresses it generates two or more Memory-write operations. In some implementations, the result of two STR instructions that write to adjacent memory might be merged into a single Memory-write operation.

Note

The data written back from a cache that is shared with other PEs might not be data that was written by the PE that performs the operation that leads to the write-back. Nevertheless, the event is counted as a write-back event for that PE.

Memory-writing instructions

Are instructions with explicit Memory write effects, including Tag-write effects. Stores and atomic instructions are counted as Memory-writing instructions.

Unless otherwise specified, this includes instructions that can have a Memory write effect, even if the instruction does not actually write to memory for some other reason. For example, a CAS or Store-exclusive that does not update memory, or a store that generates an MMU fault.

DC ZVA is counted as Memory-writing instructions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Microarchitectural event

A microarchitectural event is any event which is not architectural. That is, it will give different results for the same program on two different implementations of the Arm architecture, due to microarchitectural differences in the implementations. L1D_CACHE_REFILL is an example of a microarchitectural event.

A microarchitectural event should not be confused with an event defined by the implementation, which is referred to as an IMPLEMENTATION DEFINED event.

Note: Do not confuse Microarchitectural events and IMPLEMENTAION DEFINED events. A Microarchitectural event can be either Common or IMPLEMENTATION DEFINED. See also Common event and IMPLEMENTATION DEFINED event.

MSIZE

Memory element access size, in bits, that corresponds to a load or store instruction mnemonic suffix, where B=8, H=16, W=32 and D=64. When an instruction mnemonic does not end with B, H, W or D, the memory access size is implied by the scalar transfer register size or SIMD transfer register element size.

Operation

An instruction might create one or more microarchitectural operations (µ-ops) at any point in the execution pipeline. Depending on the event definition, the µ-ops might be counted instead of instructions. The definition of a µ-op is IMPLEMENTATION SPECIFIC. An architecture instruction might create more than one µ-op for each instruction. µ-ops might also be removed or merged in the execution stream, so an architecture instruction might create no µ-ops for an instruction. Any arbitrary translation of instructions to an equivalent sequence of µ-ops is permitted.

This means the relationship between a µ-op and an architecturally executed instruction is IMPLEMENTATION DEFINED.

Note

The architecture does not require that an implementation that generates µ-ops must count µ-ops for operations. An implementation can choose to interpret operation as instruction.

The counting of operations can indicate the workload on the PE. However, there is no requirement for operations to represent similar amounts of work, and direct comparisons between different microarchitectures are not meaningful.

Operations might be defined with reference to a particular instruction or type of instruction. In the case of operations this means operations with semantics that map to that type of instruction.

Operation speculatively executed

An Operation that is Speculatively executed.

There is no architecturally guaranteed relationship between a Speculatively executed micro-op and an architecturally executed instruction.

The results of such an operation can also be discarded, if it transpires that the operation was not required, such as following a mispredicted branch. Therefore, the architecture defines these events as *operations speculatively executed*, where appropriate.

Many PMU events count operations that are speculatively executed. This means that the PMU can observe and record the behavior of operations executed under the effects of speculation. However, the PMU must not count an event that might be used to infer values that cannot be accessed or used by the operation, as defined by Restrictions on the effects of speculation.

For example:

- An address that is not permitted to be used by the operation might be inferred by whether it is present in a cache or TLB.
- A condition code that is not permitted to be used by the operation can be inferred by whether a branch was taken or not taken.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- A predicate value that is not permitted to be used by the operation can be inferred by whether a predicate value is empty or not.

Note

In some events, operation has a more specific meaning described in the event.

Processor cycle

For a non-multithreaded implementation, this means a cycle of the processor. For a multithreaded implementation, processor cycle means each cycle of the multithreaded processor, not just those cycles for which the PE counting the event is active.

Slot

An implementation of a PE might be able to execute multiple micro-ops in a single processor cycle. The maximum number of micro-ops that can be executed might vary at different points in the execution pipeline.

To allow profiling of the utilization of the resource of the PE, an IMPLEMENTATION SPECIFIC point in the execution pipeline is chosen where the maximum number of micro-ops that can be executed is an IMPLEMENTATION DEFINED fixed value.

Each possible micro-op that can be executed at that point in a cycle is called a Slot. The maximum number of micro-ops that can be executed is defined by PMMIR.SLOTS.

Software change of the PC

All of the following are treated as a software change of the PC:

- Branch instructions.
- Memory-reading instructions that explicitly write to the PC.
- Data-processing instructions that explicitly write to the PC.
- Exception return instructions.

It is IMPLEMENTATION DEFINED whether any or all of the following are treated as software changes of the PC:

- `BRK` instructions.
- An exception generated because an instruction is UNDEFINED.
- The exception-generating instructions, `SVC` and `HVC`.
- Context synchronization barrier `ISB` instructions.

Speculatively executed

An instruction or operation that is counted by an event when it might be Speculative.

The architecture does not define the point in a pipeline where an event is counted. For some events, this means the operation or instruction is counted when the operation or instruction is Speculative. The results of such an operation or instruction might later be discarded, if it transpires that the operation was not required, such as following a mispredicted branch, or might be later resolved to be Architecturally executed.

Different groups of events might be counted at different points in the pipeline and so can have different IMPLEMENTATION DEFINED definitions of speculatively executed. Such groups share a common base type, which the event name denotes. Each of the events in the previous example is of the base type, operation speculatively executed.

For groups of events with a common base type, speculatively executed operations are all counted on the same basis, which normally means at the same point in the pipeline. It is possible to compare the counts and make meaningful observations about the program being profiled.

Taken locally

Taken locally is a qualifier that determines which instances of an exception are counted by particular PMU events. See Counting exceptions taken locally or not taken locally.

In this context, an exception that is *Taken locally* means an exception that is one of:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Taken to the current Exception level.

  Note

  This is not possible when the current Exception level is EL0.

- Taken from EL0 to EL1.
- Taken from EL0 to EL2 because the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

  Note

  An exception taken from EL0 to EL2 is not *Taken locally*.

D9.1.2 Levels of caches and TLBs

The mapping of levels of cache and TLB to the PMU events is IMPLEMENTATION DEFINED. Although CLIDR_EL1 and CLIDR define the implemented levels of cache, these are not required to correspond with the levels of cache defined for PMU events. The architecture does not provide any way of determining implemented levels of TLB. Also, many implementations include structures that provide some caching at a higher level than the level 1 caches or TLBs. Typically, these structures, that might be called Level 0 caches, or mini caches, or microcaches, are invisible to software. The IMPLEMENTATION SPECIFIC nature of cache and TLB implementations mean that, in general, PMU event counts cannot be used reliably to make direct comparisons between different implementations, and Arm recommends the following implementation guidelines:

- If L3D_CACHE events are implemented, then L2D_CACHE and if applicable L2I_CACHE events should be implemented.
- If L2D_CACHE events are implemented, then L1D_CACHE and if applicable L1I_CACHE events should be implemented.
- If L2I_CACHE events are implemented, then L1I_CACHE events should be implemented.
- Where the Last Level of cache is also the Level 3 or Level 2 unified cache, the LL_CACHE events should be implemented in preference to the L3D_CACHE or L2D_CACHE events as applicable.
- For the Level <n> cache, where <n> = 1 or <n> = 2:
  - If the Level <n> cache is unified, but the cache can disambiguate between Data and Instruction accesses to the cache, then both the L<n>D_CACHE and L<n>I_CACHE events should be implemented.
  - If the cache is unified and the cache cannot disambiguate between Data and Instruction accesses, then only the L<n>D_CACHE should be implemented, counting all accesses.

This final property is IMPLEMENTATION DEFINED.

N1_*, N2_*, N3_*, and N4_* events are defined for identifying the *non-uniform memory access* (NUMA) distance for data returned directly from caches and memory, with support for:

- Up to four NUMA distance levels.
- Up to three cache types at each level.
- Up to three memory types at each level.
- Cache snoop responses at each level.

The NUMA distance indicates the relative distance that the return data has traveled. The NUMA distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have different levels of hierarchy. The interpretation of NUMA distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system.

The mapping of NUMA distance values to LOCAL and REMOTE events is IMPLEMENTATION DEFINED.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be broadly similar for all devices in the system.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D9.1.3 Counting events from shared components

There is no architectural concept of a *shared* component. However, when a cache, a bus, or any other system component that might generate countable events is implemented, and:

- The extent of the first-order effects due to an event from that component are only applicable to a single PE, then the event is not shared.
- Otherwise, the event is shared.

Second-order effects are not considered when determining if an event is shared.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is `0b0`, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is `0b1`, the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

Example D9-2 First and second order effects of a cache miss in a multiple-PE implementation

In an implementation that consists of two PEs, each with its own L1 cache, a cache miss by one of the PEs is a first-order effect of an access to its cache. Any snoop that is performed on the L1 cache of the other PE in the implementation as a result of that cache miss is a second order effect.

Note

Shared events are inherently linked to microarchitectures and so the implementer must make an informed decision about how such events are implemented.

D9.1.4 Counting exceptions taken locally or not taken locally

Table D9-1 shows the events for exceptions taken to an Exception level using AArch64.

Table D9-1 Events for exceptions taken to an Exception level using AArch64

ESR.EC	Description	Event number and classification for exceptions	
		Taken locally	Not Taken locally
`0x00`	Unknown or uncategorized	`0x0081`, EXC_UNDEF	`0x008D`, EXC_TRAP_OTHER
`0x01`	`WF*` traps	`0x0081`, EXC_UNDEF	`0x008D`, EXC_TRAP_OTHER
`0x07`	Advanced SIMD or FP traps	`0x0081`, EXC_UNDEF	`0x008D`, EXC_TRAP_OTHER
`0x15`	AArch64 `SVC`	`0x0082`, EXC_SVC	`0x008D`, EXC_TRAP_OTHER
`0x16`	AArch64 `HVC` that is not disabled	`0x008A`, EXC_HVC	`0x008A`, EXC_HVC
`0x17`	-	-	-
`0x18`	AArch64 `MSR`, `MRS` and System instruction traps	`0x0081`, EXC_UNDEF	`0x008D`, EXC_TRAP_OTHER
`0x19`	-	-	-
`0x1E`	-	-	-
`0x1F`	-	-	-
`0x20`	Instruction Abort from below	`0x0083`, EXC_PABORT	`0x008B`, EXC_TRAP_PABORT

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ESR.EC	Description	Event number and classification for exceptions	
		Taken locally	Not Taken locally
0x21	Instruction Abort from current Exception level	0x0083, EXC_PABORT	-
0x22	PC alignment	0x0083, EXC_PABORT	0x008B, EXC_TRAP_PABORT
0x24	Data Abort from below	0x0084, EXC_DABORT	0x008C, EXC_TRAP_DABORT
0x25	Data Abort from current Exception level	0x0084, EXC_DABORT	-
0x26	SP alignment fault exception	0x0084, EXC_DABORT	0x008C, EXC_TRAP_DABORT
0x2C	AArch64 FP exception	0x0081, EXC_UNDEF	0x008D, EXC_TRAP_OTHER
0x2F	SError interrupt	0x0084, EXC_DABORT	0x008C, EXC_TRAP_DABORT
0x30	Breakpoint from below	0x0083, EXC_PABORT	0x008B, EXC_TRAP_PABORT
0x31	Breakpoint from current Exception level	0x0083, EXC_PABORT	-
0x32	Software step from below	0x0083, EXC_PABORT	0x008B, EXC_TRAP_PABORT
0x33	Software step from current Exception level	0x0083, EXC_PABORT	-
0x34	Watchpoint from below	0x0084, EXC_DABORT	0x008C, EXC_TRAP_DABORT
0x35	Watchpoint from current Exception level	0x0084, EXC_DABORT	-
0x3C	AArch64 BRK instruction	0x0083, EXC_PABORT	0x008B, EXC_TRAP_PABORT
-	IRQ interrupt	0x0086, EXC_IRQ	0x008E, EXC_TRAP_FIQ
-	FIQ interrupt	0x0087, EXC_FIQ	0x008F, EXC_TRAP_FIQ
All other values	All other exceptions	0x0081, EXC_UNDEF	0x008D, EXC_TRAP_OTHER

Note

The Glossary defines the term Taken locally, that is used in event definitions in Common microarchitectural events. See also Exception levels for more information.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D9.2 The PMU event number space and common events

In Armv8.0, the event number space is 10 bits. Armv8.1 extends the event number space, and therefore the PMEVTYPER<n>_EL0.evtCount field, to 16 bits, and is allocated as Table D9-2 shows. For more information about the entries in the Allocation column see the text that follows this table:

Table D9-2 Allocation of the PMU event number space

Event numbers	Allocation
In all versions from Armv8.0	
0x0000 - 0x003F	Common architectural and microarchitectural events.
0x0040 - 0x00BF	When FEAT_PMUv3p8 is implemented, common architectural and microarchitectural events. Previously Arm-recommended common architectural and microarchitectural events.
0x00C0 - 0x03FF	IMPLEMENTATION DEFINED events.
From Armv8.1	
0x0400 - 0x3FFF	IMPLEMENTATION DEFINED events.
0x4000 - 0x403F	Common architectural and microarchitectural events.
0x4040 - 0x40BF	When FEAT_PMUv3p8 is implemented, common architectural and microarchitectural events. Previously Arm-recommended common architectural and microarchitectural events.
0x40C0 - 0x7FFF	IMPLEMENTATION DEFINED events.
0x8000 -0x80FF	Common architectural and microarchitectural events.
0x8100 - 0xC0BF	Reserved.
0xC0C0 - 0xFFFF	IMPLEMENTATION DEFINED events.

See PMEVTYPER<n>.evtCount for details of the PE behavior when an event number for a reserved or unimplemented PMU event is written to evtCount.

Table D9-3 lists the number, mnemonic, and description of PMU events.

Table D9-3 Event index

Number	Mnemonic	Description
0x0000	SW_INCR	Instruction architecturally executed, Condition code check pass, software increment.
0x0001	L1I_CACHE_REFILL	Level 1 instruction cache refill.
0x0002	L1I_TLB_REFILL	Level 1 instruction TLB refill.
0x0003	L1D_CACHE_REFILL	Level 1 data cache refill.
0x0004	L1D_CACHE	Level 1 data cache access.
0x0005	L1D_TLB_REFILL	Level 1 data TLB refill.
0x0006	LD_RETIRED	Instruction architecturally executed, Condition code check pass, load.
0x0007	ST_RETIRED	Instruction architecturally executed, Condition code check pass, store.
0x0008	INST_RETIRED	Instruction architecturally executed.
0x0009	EXC_TAKEN	Exception taken.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x000A	EXC_RETURN	Instruction architecturally executed, Condition code check pass, exception return.
0x000B	CID_WRITE_RETIRED	Instruction architecturally executed, Condition code check pass, write to CONTEXTIDR.
0x000C	PC_WRITE_RETIRED	Instruction architecturally executed, Condition code check pass, Software change of the PC.
0x000D	BR_IMMED_RETIRED	Branch instruction architecturally executed, immediate.
0x000E	BR_RETURN_RETIRED	Branch instruction architecturally executed, procedure return, taken.
0x000F	UNALIGNED_LDST_RETIRED	Instruction architecturally executed, Condition code check pass, unaligned load or store.
0x0010	BR_MIS_PRED	Branch instruction speculatively executed, mispredicted or not predicted.
0x0011	CPU_CYCLES	Cycle.
0x0012	BR_PRED	Predictable branch instruction speculatively executed.
0x0013	MEM_ACCESS	Data memory access.
0x0014	L1I_CACHE	Level 1 instruction cache access.
0x0015	L1D_CACHE_WB	Level 1 data cache write-back.
0x0016	L2D_CACHE	Level 2 data cache access.
0x0017	L2D_CACHE_REFILL	Level 2 data cache refill.
0x0018	L2D_CACHE_WB	Level 2 data cache write-back.
0x0019	BUS_ACCESS	Bus access.
0x001A	MEMORY_ERROR	Local memory error.
0x001B	INST_SPEC	Operation speculatively executed.
0x001C	TTBR_WRITE_RETIRED	Instruction architecturally executed, Condition code check pass, write to TTBR.
0x001D	BUS_CYCLES	Bus cycle.
0x001E	CHAIN	Chain a pair of event counters.
0x001F	L1D_CACHE_ALLOCATE	Level 1 data cache allocation without refill.
0x0020	L2D_CACHE_ALLOCATE	Level 2 data cache allocation without refill.
0x0021	BR_RETIRED	Instruction architecturally executed, branch.
0x0022	BR_MIS_PRED_RETIRED	Branch instruction architecturally executed, mispredicted.
0x0023	STALL_FRONTEND	No operation sent for execution due to the frontend.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x0024	STALL_BACKEND	No operation sent for execution due to the backend.
0x0025	L1D_TLB	Level 1 data TLB access.
0x0026	L1I_TLB	Level 1 instruction TLB access.
0x0027	L2I_CACHE	Level 2 instruction cache access.
0x0028	L2I_CACHE_REFILL	Level 2 instruction cache refill.
0x0029	L3D_CACHE_ALLOCATE	Level 3 data cache allocation without refill.
0x002A	L3D_CACHE_REFILL	Level 3 data cache refill.
0x002B	L3D_CACHE	Level 3 data cache access.
0x002C	L3D_CACHE_WB	Level 3 data cache write-back.
0x002D	L2D_TLB_REFILL	Level 2 data TLB refill.
0x002E	L2I_TLB_REFILL	Level 2 instruction TLB refill.
0x002F	L2D_TLB	Level 2 data TLB access.
0x0030	L2I_TLB	Level 2 instruction TLB access.
0x0031	REMOTE_ACCESS	Access to a remote device.
0x0032	LL_CACHE	Last level cache access.
0x0033	LL_CACHE_MISS	Last level cache miss.
0x0034	DTLB_WALK	Data TLB access with at least one translation table walk.
0x0035	ITLB_WALK	Instruction TLB access with at least one translation table walk.
0x0036	LL_CACHE_RD	Last level cache access, read.
0x0037	LL_CACHE_MISS_RD	Last level cache miss, read.
0x0038	REMOTE_ACCESS_RD	Access to a remote device, read.
0x0039	L1D_CACHE_LMISS_RD	Level 1 data cache long-latency read miss.
0x003A	OP_RETIRED	Micro-operation architecturally executed.
0x003B	OP_SPEC	Micro-operation speculatively executed.
0x003C	STALL	No operation sent for execution.
0x003D	STALL_SLOT_BACKEND	No operation sent for execution on a Slot due to the backend.
0x003E	STALL_SLOT_FRONTEND	No operation sent for execution on a Slot due to the frontend.
0x003F	STALL_SLOT	No operation sent for execution on a Slot.
0x0040	L1D_CACHE_RD	Level 1 data cache access, read.
0x0041	L1D_CACHE_WR	Level 1 data cache access, write.
0x0042	L1D_CACHE_REFILL_RD	Level 1 data cache refill, read.
0x0043	L1D_CACHE_REFILL_WR	Level 1 data cache refill, write.
0x0044	L1D_CACHE_REFILL_INNER	Level 1 data cache refill, inner.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x0045	L1D_CACHE_REFILL_OUTER	Level 1 data cache refill, outer.
0x0046	L1D_CACHE_WB_VICTIM	Level 1 data cache write-back, victim.
0x0047	L1D_CACHE_WB_CLEAN	Level 1 data cache write-back, cleaning and coherency.
0x0048	L1D_CACHE_INVAL	Level 1 data cache invalidate.
0x004C	L1D_TLB_REFILL_RD	Level 1 data TLB refill, read.
0x004D	L1D_TLB_REFILL_WR	Level 1 data TLB refill, write.
0x004E	L1D_TLB_RD	Level 1 data TLB access, read.
0x004F	L1D_TLB_WR	Level 1 data TLB access, write.
0x0050	L2D_CACHE_RD	Level 2 data cache access, read.
0x0051	L2D_CACHE_WR	Level 2 data cache access, write.
0x0052	L2D_CACHE_REFILL_RD	Level 2 data cache refill, read.
0x0053	L2D_CACHE_REFILL_WR	Level 2 data cache refill, write.
0x0056	L2D_CACHE_WB_VICTIM	Level 2 data cache write-back, victim.
0x0057	L2D_CACHE_WB_CLEAN	Level 2 data cache write-back, cleaning and coherency.
0x0058	L2D_CACHE_INVAL	Level 2 data cache invalidate.
0x005C	L2D_TLB_REFILL_RD	Level 2 data TLB refill, read.
0x005D	L2D_TLB_REFILL_WR	Level 2 data TLB refill, write.
0x005E	L2D_TLB_RD	Level 2 data TLB access, read.
0x005F	L2D_TLB_WR	Level 2 data TLB access, write.
0x0060	BUS_ACCESS_RD	Bus access, read.
0x0061	BUS_ACCESS_WR	Bus access, write.
0x0062	BUS_ACCESS_SHARED	Bus access, Normal, Cacheable, Shareable.
0x0063	BUS_ACCESS_NOT_SHARED	Bus access, not Normal, Cacheable, Shareable.
0x0064	BUS_ACCESS_NORMAL	Bus access, normal.
0x0065	BUS_ACCESS_PERIPH	Bus access, peripheral.
0x0066	MEM_ACCESS_RD	Data memory access, read.
0x0067	MEM_ACCESS_WR	Data memory access, write.
0x0068	UNALIGNED_LD_SPEC	Unaligned access, read.
0x0069	UNALIGNED_ST_SPEC	Unaligned access, write.
0x006A	UNALIGNED_LDST_SPEC	Unaligned access.
0x006C	LDREX_SPEC	Exclusive operation speculatively executed, Load-Exclusive.
0x006D	STREX_PASS_SPEC	Exclusive operation speculatively executed, Store-Exclusive pass.
0x006E	STREX_FAIL_SPEC	Exclusive operation speculatively executed, Store-Exclusive fail.
0x006F	STREX_SPEC	Exclusive operation speculatively executed, Store-Exclusive.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x0070	LD_SPEC	Operation speculatively executed, load.
0x0071	ST_SPEC	Operation speculatively executed, store.
0x0072	LDST_SPEC	Operation speculatively executed, load or store.
0x0073	DP_SPEC	Operation speculatively executed, integer data processing.
0x0074	ASE_SPEC	Operation speculatively executed, Advanced SIMD data processing.
0x0075	VFP_SPEC	Operation speculatively executed, scalar floating-point data processing.
0x0076	PC_WRITE_SPEC	Operation speculatively executed, Software change of the PC.
0x0077	CRYPTO_SPEC	Operation speculatively executed, cryptographic data processing.
0x0078	BR_IMMED_SPEC	Branch speculatively executed, immediate branch.
0x0079	BR_RETURN_SPEC	Branch speculatively executed, procedure return.
0x007A	BR_INDIRECT_SPEC	Branch speculatively executed, indirect branch.
0x007C	ISB_SPEC	Barrier speculatively executed, ISB.
0x007D	DSB_SPEC	Barrier speculatively executed, DSB.
0x007E	DMB_SPEC	Barrier speculatively executed, DMB.
0x007F	CSDB_SPEC	Barrier speculatively executed, CSDB.
0x0081	EXC_UNDEF	Exception taken, other synchronous.
0x0082	EXC_SVC	Exception taken, Supervisor Call.
0x0083	EXC_PABORT	Exception taken, Instruction Abort.
0x0084	EXC_DABORT	Exception taken, Data Abort or SError.
0x0086	EXC_IRQ	Exception taken, IRQ.
0x0087	EXC_FIQ	Exception taken, FIQ.
0x0088	EXC_SMC	Exception taken, Secure Monitor Call.
0x008A	EXC_HVC	Exception taken, Hypervisor Call.
0x008B	EXC_TRAP_PABORT	Exception taken, Instruction Abort not Taken locally.
0x008C	EXC_TRAP_DABORT	Exception taken, Data Abort or SError not Taken locally.
0x008D	EXC_TRAP_OTHER	Exception taken, other traps not Taken locally.
0x008E	EXC_TRAP_IRQ	Exception taken, IRQ not Taken locally.
0x008F	EXC_TRAP_FIQ	Exception taken, FIQ not Taken locally.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x0090	RC_LD_SPEC	Release consistency operation speculatively executed, Load-Acquire.
0x0091	RC_ST_SPEC	Release consistency operation speculatively executed, Store-Release.
0x00A0	L3D_CACHE_RD	Level 3 data cache access, read.
0x00A1	L3D_CACHE_WR	Level 3 data cache access, write.
0x00A2	L3D_CACHE_REFILL_RD	Level 3 data cache refill, read.
0x00A3	L3D_CACHE_REFILL_WR	Level 3 data cache refill, write.
0x00A6	L3D_CACHE_WB_VICTIM	Level 3 data cache write-back, victim.
0x00A7	L3D_CACHE_WB_CLEAN	Level 3 data cache write-back, cleaning and coherency.
0x00A8	L3D_CACHE_INVAL	Level 3 data cache invalidate.
0x4000	SAMPLE_POP	Statistical Profiling sample population.
0x4001	SAMPLE_FEED	Statistical Profiling sample taken.
0x4002	SAMPLE_FILTRATE	Statistical Profiling sample taken and not removed by filtering.
0x4003	SAMPLE_COLLISION	Statistical Profiling sample collided with previous sample.
0x4004	CNT_CYCLES	Constant frequency cycles.
0x4005	STALL_BACKEND_MEM	Memory stall cycles.
0x4006	L1I_CACHE_LMISS	Level 1 instruction cache long-latency miss.
0x4009	L2D_CACHE_LMISS_RD	Level 2 data cache long-latency read miss.
0x400A	L2I_CACHE_LMISS	Level 2 instruction cache long-latency miss.
0x400B	L3D_CACHE_LMISS_RD	Level 3 data cache long-latency read miss.
0x400C	TRB_WRAP	Trace buffer current write pointer wrapped.
0x4010	TRCEXTOUT0	Trace unit external output 0.
0x4011	TRCEXTOUT1	Trace unit external output 1.
0x4012	TRCEXTOUT2	Trace unit external output 2.
0x4013	TRCEXTOUT3	Trace unit external output 3.
0x4018	CTI_TRIGOUT4	Cross-trigger Interface output trigger 4.
0x4019	CTI_TRIGOUT5	Cross-trigger Interface output trigger 5.
0x401A	CTI_TRIGOUT6	Cross-trigger Interface output trigger 6.
0x401B	CTI_TRIGOUT7	Cross-trigger Interface output trigger 7.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x4020	LDST_ALIGN_LAT	Access with additional latency from alignment.
0x4021	LD_ALIGN_LAT	Load with additional latency from alignment.
0x4022	ST_ALIGN_LAT	Store with additional latency from alignment.
0x4030	TSTART_RETIRED	Instruction architecturally executed, outer transaction TSTART.
0x4031	TCOMMIT_RETIRED	Instruction architecturally executed, outer transaction TCOMMIT.
0x4032	TME_TRANSACTION_FAILED	Transaction failed or was canceled.
0x4034	TME_INST_RETIRED_COMMITTED	Instruction architecturally executed, in a committed transaction.
0x4035	TME_CPU_CYCLES_COMMITTED	PE cycle, in a committed transaction.
0x4038	TME_FAILURE_CNCL	Transaction failed with CNCL cause.
0x4039	TME_FAILURE_NEST	Transaction failed with NEST cause.
0x403A	TME_FAILURE_ERR	Transaction failed with ERR cause.
0x403B	TME_FAILURE_IMP	Transaction failed with IMP cause.
0x403C	TME_FAILURE_MEM	Transaction failed with MEM cause.
0x403D	TME_FAILURE_SIZE	Transaction failed with SIZE cause.
0x403E	TME_FAILURE_TLBI	Transaction failed due to execution of TLBI by another PE.
0x403F	TME_FAILURE_WSET	Transaction failed due to transactional write set limit overflow.
0x8000	SIMD_INST_RETIRED	Instruction architecturally executed, SIMD.
0x8001	ASE_INST_RETIRED	Instruction architecturally executed, Advanced SIMD.
0x8003	ASE_SVE_INST_RETIRED	Instruction architecturally executed, Advanced SIMD or SVE.
0x8004	SIMD_INST_SPEC	Operation speculatively executed, SIMD.
0x8005	ASE_INST_SPEC	Operation speculatively executed, Advanced SIMD.
0x8007	ASE_SVE_INST_SPEC	Operation speculatively executed, Advanced SIMD or SVE.
0x8008	UOP_SPEC	Microarchitectural operation speculatively executed.
0x8009	ASE_UOP_SPEC	Microarchitectural operation speculatively executed, Advanced SIMD.
0x800B	ASE_SVE_UOP_SPEC	Microarchitectural operation speculatively executed, Advanced SIMD or SVE.
0x800C	SIMD_UOP_SPEC	Microarchitectural operation speculatively executed, SIMD.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x8010	FP_SPEC	Floating-point operation speculatively executed, including SIMD.
0x8011	ASE_FP_SPEC	Floating-point operation speculatively executed, Advanced SIMD.
0x8013	ASE_SVE_FP_SPEC	Floating-point operation speculatively executed, Advanced SIMD or SVE.
0x8014	FP_HP_SPEC	Floating-point operation speculatively executed, half-precision.
0x8015	ASE_FP_HP_SPEC	Floating-point operation speculatively executed, Advanced SIMD half-precision, data-processing.
0x8017	ASE_SVE_FP_HP_SPEC	Floating-point operation speculatively executed, Advanced SIMD or SVE half-precision.
0x8018	FP_SP_SPEC	Floating-point operation speculatively executed, single-precision.
0x8019	ASE_FP_SP_SPEC	Floating-point operation speculatively executed, Advanced SIMD single-precision, data-processing.
0x801B	ASE_SVE_FP_SP_SPEC	Floating-point operation speculatively executed, Advanced SIMD or SVE single-precision, data-processing.
0x801C	FP_DP_SPEC	Floating-point operation speculatively executed, double-precision, data-processing.
0x801D	ASE_FP_DP_SPEC	Floating-point operation speculatively executed, Advanced SIMD double-precision, data-processing.
0x801F	ASE_SVE_FP_DP_SPEC	Floating-point operation speculatively executed, Advanced SIMD or SVE double-precision, data-processing.
0x8020	FP_DIV_SPEC	Floating-point operation speculatively executed, divide.
0x8021	ASE_FP_DIV_SPEC	Floating-point operation speculatively executed, Advanced SIMD divide.
0x8023	ASE_SVE_FP_DIV_SPEC	Floating-point operation speculatively executed, Advanced SIMD or SVE divide.
0x8024	FP_SQRT_SPEC	Floating-point operation speculatively executed, square root.
0x8025	ASE_FP_SQRT_SPEC	Floating-point operation speculatively executed, Advanced SIMD square root.
0x8027	ASE_SVE_FP_SQRT_SPEC	Floating-point operation speculatively executed, Advanced SIMD or SVE square-root.
0x8028	FP_FMA_SPEC	Floating-point operation speculatively executed, FMA.
0x8029	ASE_FP_FMA_SPEC	Floating-point operation speculatively executed, Advanced SIMD FMA.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x802B	ASE_SVE_FP_FMA_SPEC	Floating-point operation speculatively executed, Advanced SIMD or SVE FMA.
0x802C	FP_MUL_SPEC	Floating-point operation speculatively executed, multiply.
0x802D	ASE_FP_MUL_SPEC	Floating-point operation speculatively executed, Advanced SIMD multiply.
0x802F	ASE_SVE_FP_MUL_SPEC	Floating-point operation speculatively executed, Advanced SIMD or SVE multiply.
0x8030	FP_ADDSUB_SPEC	Floating-point operation speculatively executed, add or subtract.
0x8031	ASE_FP_ADDSUB_SPEC	Floating-point operation speculatively executed, Advanced SIMD add or subtract.
0x8033	ASE_SVE_FP_ADDSUB_SPEC	Floating-point operation speculatively executed, Advanced SIMD or SVE add or subtract .
0x8034	FP_RECPE_SPEC	Floating-point operation speculatively executed, reciprocal estimate.
0x8035	ASE_FP_RECPE_SPEC	Floating-point operation speculatively executed, Advanced SIMD reciprocal estimate.
0x8037	ASE_SVE_FP_RECPE_SPEC	Floating-point operation speculatively executed, Advanced SIMD or SVE reciprocal estimate.
0x8038	FP_CVT_SPEC	Floating-point operation speculatively executed, convert.
0x8039	ASE_FP_CVT_SPEC	Floating-point operation speculatively executed, Advanced SIMD convert.
0x803B	ASE_SVE_FP_CVT_SPEC	Floating-point operation speculatively executed, Advanced SIMD or SVE convert.
0x803D	ASE_FP_PREDUCE_SPEC	Floating-point operation speculatively executed, Advanced SIMD pairwise add step or pairwise reduce step.
0x803F	ASE_SVE_FP_VREDUCE_SPEC	Floating-point operation speculatively executed, Advanced SIMD or SVE vector reduction.
0x8040	INT_SPEC	Integer operation speculatively executed.
0x8041	ASE_INT_SPEC	Integer operation speculatively executed, Advanced SIMD.
0x8043	ASE_SVE_INT_SPEC	Integer operation speculatively executed, Advanced SIMD or SVE.
0x8044	INT_DIV_SPEC	Integer operation speculatively executed, divide.
0x8045	INT_DIV64_SPEC	Integer operation speculatively executed, 64-bit divide.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x8048	INT_MUL_SPEC	Integer operation speculatively executed, multiply.
0x8049	ASE_INT_MUL_SPEC	Integer operation speculatively executed, Advanced SIMD multiply.
0x804B	ASE_SVE_INT_MUL_SPEC	Integer operation speculatively executed, Advanced SIMD or SVE multiply.
0x804C	INT_MUL64_SPEC	Integer operation speculatively executed, 64×64 multiply.
0x804E	INT_MULH64_SPEC	Integer operation speculatively executed, 64×64 multiply returning high part.
0x8053	ASE_SVE_FP_BF16_SPEC	Floating-point operation speculatively executed, SVE or Advanced SIMD BFloat16, data-processing.
0x8056	SVE_SPEC	Operation speculatively executed, SVE data processing.
0x8057	ASE_SVE_SPEC	Operation speculatively executed, Advanced SIMD data processing or SVE data processing.
0x8058	NONFP_SPEC	Non-floating-point operation speculatively executed.
0x8059	ASE_NONFP_SPEC	Non-floating-point operation speculatively executed, Advanced SIMD.
0x805B	ASE_SVE_NONFP_SPEC	Non-floating-point operation speculatively executed, Advanced SIMD or SVE.
0x805D	ASE_INT_VREDUCE_SPEC	Integer operation speculatively executed, Advanced SIMD reduction.
0x805F	ASE_SVE_INT_VREDUCE_SPEC	Integer operation speculatively executed, Advanced SIMD or SVE reduction.
0x8084	ASE_SVE_LDST_SPEC	Operation speculatively executed, Advanced SIMD or SVE load or store.
0x8085	ASE_SVE_LD_SPEC	Operation speculatively executed, Advanced SIMD or SVE load.
0x8086	ASE_SVE_ST_SPEC	Operation speculatively executed, Advanced SIMD or SVE store.
0x8087	PRF_SPEC	Operation speculatively executed, prefetch.
0x8088	BASE_LDST_REG_SPEC	Operation speculatively executed, general-purpose register load, store, or prefetch.
0x8089	BASE_LD_REG_SPEC	Operation speculatively executed, general-purpose register load.
0x808A	BASE_ST_REG_SPEC	Operation speculatively executed, general-purpose register store.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x808B	BASE_PRF_SPEC	Operation speculatively executed, general-purpose register prefetch.
0x808C	FPASE_LDST_REG_SPEC	Operation speculatively executed, SIMD&FP register load or store.
0x808D	FPASE_LD_REG_SPEC	Operation speculatively executed, SIMD&FP register load.
0x808E	FPASE_ST_REG_SPEC	Operation speculatively executed, SIMD&FP register store.
0x80A4	ASE_SVE_LDST_MULTI_SPEC	Operation speculatively executed, Advanced SIMD or SVE contiguous load or store multiple vector.
0x80A5	ASE_SVE_LD_MULTI_SPEC	Operation speculatively executed, Advanced SIMD or SVE contiguous load multiple vector.
0x80A6	ASE_SVE_ST_MULTI_SPEC	Operation speculatively executed, Advanced SIMD or SVE contiguous store multiple vector.
0x80B4	ASE_SVE_UNALIGNED_LDST_SPEC	Advanced SIMD or SVE unaligned access.
0x80B5	ASE_SVE_UNALIGNED_LD_SPEC	Advanced SIMD or SVE unaligned read.
0x80B6	ASE_SVE_UNALIGNED_ST_SPEC	Advanced SIMD or SVE unaligned write.
0x80B8	ASE_SVE_UNALIGNED_CONTIG_LDST_SPEC	Advanced SIMD or SVE unaligned contiguous access.
0x80B9	ASE_SVE_UNALIGNED_CONTIG_LD_SPEC	Advanced SIMD or SVE unaligned contiguous read.
0x80BA	ASE_SVE_UNALIGNED_CONTIG_ST_SPEC	Advanced SIMD or SVE unaligned contiguous write.
0x80C1	FP_FIXED_OPS_SPEC	Non-scalable element arithmetic operations speculatively executed, floating-point.
0x80C3	FP_HP_FIXED_OPS_SPEC	Non-scalable element arithmetic operations speculatively executed, largest type is half-precision floating-point.
0x80C5	FP_SP_FIXED_OPS_SPEC	Non-scalable element arithmetic operations speculatively executed, largest type is single-precision floating-point.
0x80C7	FP_DP_FIXED_OPS_SPEC	Non-scalable element arithmetic operations speculatively executed, largest type is double-precision floating-point.
0x80C9	INT_FIXED_OPS_SPEC	Non-scalable element arithmetic operations speculatively executed, integer.
0x80CB	LDST_FIXED_OPS_SPEC	Non-scalable load or store element operations speculatively executed.
0x80CD	LD_FIXED_OPS_SPEC	Non-scalable load element operations speculatively executed.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x80CF	ST_FIXED_OPS_SPEC	Non-scalable store element operations speculatively executed.
0x80DB	LDST_FIXED_BYTES_SPEC	Non-scalable load and store bytes speculatively executed.
0x80DD	LD_FIXED_BYTES_SPEC	Non-scalable load bytes speculatively executed.
0x80DF	ST_FIXED_BYTES_SPEC	Non-scalable store bytes speculatively executed.
0x80E1	ASE_INT8_SPEC	Integer operation speculatively executed, Advanced SIMD 8-bit.
0x80E3	ASE_SVE_INT8_SPEC	Integer operation speculatively executed, Advanced SIMD or SVE 8-bit.
0x80E5	ASE_INT16_SPEC	Integer operation speculatively executed, Advanced SIMD 16-bit.
0x80E7	ASE_SVE_INT16_SPEC	Integer operation speculatively executed, Advanced SIMD or SVE 16-bit.
0x80E9	ASE_INT32_SPEC	Integer operation speculatively executed, Advanced SIMD 32-bit.
0x80EB	ASE_SVE_INT32_SPEC	Integer operation speculatively executed, Advanced SIMD or SVE 32-bit.
0x80ED	ASE_INT64_SPEC	Integer operation speculatively executed, Advanced SIMD 64-bit.
0x80EF	ASE_SVE_INT64_SPEC	Integer operation speculatively executed, Advanced SIMD or SVE 64-bit.
0x80F1	ASE_FP_DOT_SPEC	Floating-point operation speculatively executed, Advanced SIMD dot-product.
0x80F3	ASE_SVE_FP_DOT_SPEC	Floating-point operation speculatively executed, Advanced SIMD or SVE dot-product.
0x80F5	ASE_FP_MMLA_SPEC	Floating-point operation speculatively executed, Advanced SIMD matrix multiply.
0x80F7	ASE_SVE_FP_MMLA_SPEC	Floating-point operation speculatively executed, Advanced SIMD or SVE matrix multiply.
0x80F9	ASE_INT_DOT_SPEC	Operation speculatively executed, Advanced SIMD integer dot-product.
0x80FB	ASE_SVE_INT_DOT_SPEC	Integer operation speculatively executed, Advanced SIMD or SVE dot-product.
0x80FD	ASE_INT_MMLA_SPEC	Integer operation speculatively executed, Advanced SIMD matrix multiply.
0x80FF	ASE_SVE_INT_MMLA_SPEC	Integer operation speculatively executed, Advanced SIMD or SVE matrix multiply.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x8107	BR_SKIP_RETIRED	Branch instruction architecturally executed, not taken.
0x8108	BR_IMMED_TAKEN_RETIRED	Branch instruction architecturally executed, immediate, taken.
0x810C	BR_INDNR_TAKEN_RETIRED	Branch instruction architecturally executed, indirect excluding procedure return, taken.
0x8110	BR_IMMED_PRED_RETIRED	Branch instruction architecturally executed, predicted immediate.
0x8111	BR_IMMED_MIS_PRED_RETIRED	Branch instruction architecturally executed, mispredicted immediate.
0x8112	BR_IND_PRED_RETIRED	Branch instruction architecturally executed, predicted indirect.
0x8113	BR_IND_MIS_PRED_RETIRED	Branch instruction architecturally executed, mispredicted indirect.
0x8114	BR_RETURN_PRED_RETIRED	Branch instruction architecturally executed, predicted procedure return.
0x8115	BR_RETURN_MIS_PRED_RETIRED	Branch instruction architecturally executed, mispredicted procedure return.
0x8116	BR_INDNR_PRED_RETIRED	Branch instruction architecturally executed, predicted indirect excluding procedure return.
0x8117	BR_INDNR_MIS_PRED_RETIRED	Branch instruction architecturally executed, mispredicted indirect excluding procedure return.
0x8118	BR_TAKEN_PRED_RETIRED	Branch instruction architecturally executed, predicted branch, taken.
0x8119	BR_TAKEN_MIS_PRED_RETIRED	Branch instruction architecturally executed, mispredicted branch, taken.
0x811A	BR_SKIP_PRED_RETIRED	Branch instruction architecturally executed, predicted branch, not taken.
0x811B	BR_SKIP_MIS_PRED_RETIRED	Branch instruction architecturally executed, mispredicted branch, not taken.
0x811C	BR_PRED_RETIRED	Branch instruction architecturally executed, predicted branch.
0x811D	BR_IND_RETIRED	Instruction architecturally executed, indirect branch.
0x811F	BRB_FILTRATE	Branch Record captured.
0x8120	INST_FETCH_PERCYC	Instruction fetches in progress.
0x8121	MEM_ACCESS_RD_PERCYC	Data memory reads in progress.
0x8122	SAMPLE_FEED_DS	Statistical Profiling sample taken, selected Data Source.
0x8123	SAMPLE_BUFFER_FULL	Profiling Buffer full.
0x8124	INST_FETCH	Instruction memory access.
0x8125	BUS_REQ_RD_PERCYC	Bus read transactions in progress.
0x8126	BUS_REQ_WR_PERCYC	Bus write transactions in progress.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x8127	PMU_SNAPSHOT	Successful PMU capture event.
0x8128	DTLB_WALK_PERCYC	Data translation table walks in progress.
0x8129	ITLB_WALK_PERCYC	Instruction translation table walks in progress.
0x8130	L1D_TLB_RW	Level 1 data TLB access, demand access.
0x8131	L1I_TLB_RD	Level 1 instruction TLB access, demand access.
0x8132	L1D_TLB_PRFM	Level 1 data TLB access, software prefetch.
0x8133	L1I_TLB_PRFM	Level 1 instruction TLB access, software prefetch.
0x8134	DTLB_HWUPD	Data TLB hardware update of translation table.
0x8135	ITLB_HWUPD	Instruction TLB hardware update of translation table.
0x8136	DTLB_STEP	Data TLB translation table walk, step.
0x8137	ITLB_STEP	Instruction TLB translation table walk, step.
0x8138	DTLB_WALK_LARGE	Data TLB large page translation table walk.
0x8139	ITLB_WALK_LARGE	Instruction TLB large page translation table walk.
0x813A	DTLB_WALK_SMALL	Data TLB small page translation table walk.
0x813B	ITLB_WALK_SMALL	Instruction TLB small page translation table walk.
0x813C	DTLB_WALK_RW	Data TLB demand access, with at least one translation table walk.
0x813D	ITLB_WALK_RD	Instruction TLB demand access, with at least one translation table walk.
0x813E	DTLB_WALK_PRFM	Data TLB software prefetch, with at least one translation table walk.
0x813F	ITLB_WALK_PRFM	Instruction TLB software prefetch, with at least one translation table walk.
0x8140	L1D_CACHE_RW	Level 1 data cache demand access.
0x8141	L1I_CACHE_RD	Level 1 instruction cache demand fetch.
0x8142	L1D_CACHE_PRFM	Level 1 data cache software prefetch.
0x8143	L1I_CACHE_PRFM	Level 1 instruction cache software prefetch.
0x8144	L1D_CACHE_MISS	Level 1 data cache demand access miss.
0x8145	L1I_CACHE_HWPRF	Level 1 instruction cache hardware prefetch.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x8146	L1D_CACHE_REFILL_PRFM	Level 1 data cache refill, software prefetch.
0x8147	L1I_CACHE_REFILL_PRFM	Level 1 instruction cache refill, software prefetch.
0x8148	L2D_CACHE_RW	Level 2 data cache demand access.
0x8149	L2I_CACHE_RD	Level 2 instruction cache demand fetch.
0x814A	L2D_CACHE_PRFM	Level 2 data cache software prefetch.
0x814B	L2I_CACHE_PRFM	Level 2 instruction cache software prefetch.
0x814C	L2D_CACHE_MISS	Level 2 data cache demand access miss.
0x814D	L2I_CACHE_HWPRF	Level 2 instruction cache hardware prefetch.
0x814E	L2D_CACHE_REFILL_PRFM	Level 2 data cache refill, software prefetch.
0x814F	L2I_CACHE_REFILL_PRFM	Level 2 instruction cache refill, software prefetch.
0x8150	L3D_CACHE_RW	Level 3 data cache demand access.
0x8151	L3D_CACHE_PRFM	Level 3 data cache software prefetch.
0x8152	L3D_CACHE_MISS	Level 3 data cache demand access miss.
0x8153	L3D_CACHE_REFILL_PRFM	Level 3 data cache refill, software prefetch.
0x8154	L1D_CACHE_HWPRF	Level 1 data cache hardware prefetch.
0x8155	L2D_CACHE_HWPRF	Level 2 data cache hardware prefetch.
0x8156	L3D_CACHE_HWPRF	Level 3 data cache hardware prefetch.
0x8157	LL_CACHE_HWPRF	Last level cache hardware prefetch.
0x8158	STALL_FRONTEND_MEMBOUND	Frontend stall cycles, memory bound.
0x8159	STALL_FRONTEND_L1I	Frontend stall cycles, level 1 instruction cache.
0x815A	STALL_FRONTEND_L2I	Frontend stall cycles, level 2 instruction cache.
0x815B	STALL_FRONTEND_MEM	Frontend stall cycles, last level PE cache or memory.
0x815C	STALL_FRONTEND_TLB	Frontend stall cycles, TLB.
0x8160	STALL_FRONTEND_CPUBOUND	Frontend stall cycles, processor bound.
0x8161	STALL_FRONTEND_FLOW	Frontend stall cycles, flow control.
0x8162	STALL_FRONTEND_FLUSH	Frontend stall cycles, flush recovery.
0x8163	STALL_FRONTEND_RENAME	Frontend stall cycles, rename full.
0x8164	STALL_BACKEND_MEMBOUND	Backend stall cycles, memory bound.
0x8165	STALL_BACKEND_L1D	Backend stall cycles, level 1 data cache.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x8166	STALL_BACKEND_L2D	Backend stall cycles, level 2 data cache.
0x8167	STALL_BACKEND_TLB	Backend stall cycles, TLB.
0x8168	STALL_BACKEND_ST	Backend stall cycles, store.
0x816A	STALL_BACKEND_CPUBOUND	Backend stall cycles, processor bound.
0x816B	STALL_BACKEND_BUSY	Backend stall cycles, backend busy.
0x816C	STALL_BACKEND_ILOCK	Backend stall cycles, input dependency.
0x816D	STALL_BACKEND_RENAME	Backend stall cycles, rename full.
0x816E	STALL_BACKEND_ATOMIC	Backend stall cycles, atomic operation.
0x8170	CAS_NEAR_FAIL	Atomic memory Operation speculatively executed, Compare and Swap fail.
0x8171	CAS_NEAR_PASS	Atomic memory Operation speculatively executed, Compare and Swap pass.
0x8172	CAS_NEAR_SPEC	Atomic memory Operation speculatively executed, Compare and Swap near.
0x8173	CAS_FAR_SPEC	Atomic memory Operation speculatively executed, Compare and Swap far.
0x8174	CAS_SPEC	Atomic memory Operation speculatively executed, Compare and Swap.
0x8175	LSE_LD_SPEC	Atomic memory Operation speculatively executed, load.
0x8176	LSE_ST_SPEC	Atomic memory Operation speculatively executed, store.
0x8177	LSE_LDST_SPEC	Atomic memory Operation speculatively executed, load or store.
0x8178	REMOTE_ACCESS_WR	Access to a remote device, write.
0x8179	BRNL_INDNR_TAKEN_RETIRED	Branch instruction architecturally executed, indirect branch without link excluding procedure return, taken.
0x817A	BL_TAKEN_RETIRED	Branch instruction architecturally executed, branch with link, taken.
0x817B	BRNL_TAKEN_RETIRED	Branch instruction architecturally executed, branch without link, taken.
0x817C	BL_IND_TAKEN_RETIRED	Branch instruction architecturally executed, indirect branch with link, taken.
0x817D	BRNL_IND_TAKEN_RETIRED	Branch instruction architecturally executed, indirect branch without link, taken.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x817E	BL_IMMED_TAKEN_RETIRED	Branch instruction architecturally executed, direct branch with link, taken.
0x817F	BRNL_IMMED_TAKEN_RETIRED	Branch instruction architecturally executed, direct branch without link, taken.
0x8180	BR_UNCOND_RETIRED	Branch instruction architecturally executed, unconditional branch.
0x8181	BR_COND_RETIRED	Branch instruction architecturally executed, conditional branch.
0x8182	BR_COND_TAKEN_RETIRED	Branch instruction architecturally executed, conditional branch, taken.
0x8186	UOP_RETIRED	Micro-operation architecturally executed.
0x8188	DTLB_WALK_BLOCK	Data TLB block translation table walk.
0x8189	ITLB_WALK_BLOCK	Instruction TLB block translation table walk.
0x818A	DTLB_WALK_PAGE	Data TLB page translation table walk.
0x818B	ITLB_WALK_PAGE	Instruction TLB page translation table walk.
0x818D	BUS_REQ_RD	Bus request, read.
0x818E	BUS_REQ_WR	Bus request, write.
0x818F	BUS_REQ	Bus request.
0x8190	ISNP_HIT_RD	Snoop hit, demand instruction fetch.
0x8191	ISNP_HIT_NEAR_RD	Snoop hit in near local cache, demand instruction fetch.
0x8192	ISNP_HIT_FAR_RD	Snoop hit in far local cache, demand instruction fetch.
0x8193	ISNP_HIT_REMOTE_RD	Snoop hit in remote cache, demand instruction fetch.
0x8194	DSNP_HIT_RD	Snoop hit, demand data read.
0x8195	DSNP_HIT_NEAR_RD	Snoop hit in near local cache, demand data read.
0x8196	DSNP_HIT_FAR_RD	Snoop hit in far local cache, demand data read.
0x8197	DSNP_HIT_REMOTE_RD	Snoop hit in remote cache, demand data read.
0x8198	DSNP_HIT_WR	Snoop hit, demand data write.
0x8199	DSNP_HIT_NEAR_WR	Snoop hit in near local cache, demand data write.
0x819A	DSNP_HIT_FAR_WR	Snoop hit in far local cache, demand data write.
0x819B	DSNP_HIT_REMOTE_WR	Snoop hit in remote cache, demand data write.
0x819C	DSNP_HIT_RW	Snoop hit, demand data access.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x819D	DSNP_HIT_NEAR_RW	Snoop hit in near local cache, demand data access.
0x819E	DSNP_HIT_FAR_RW	Snoop hit in far local cache, demand data access.
0x819F	DSNP_HIT_REMOTE_RW	Snoop hit in remote cache, demand data access.
0x81A0	DSNP_HIT_PRFM	Snoop hit, software data prefetch.
0x81A1	DSNP_HIT_NEAR_PRFM	Snoop hit in near local cache, software data prefetch.
0x81A2	DSNP_HIT_FAR_PRFM	Snoop hit in far local cache, software data prefetch.
0x81A3	DSNP_HIT_REMOTE_PRFM	Snoop hit in remote cache, software data prefetch.
0x81A4	DSNP_HIT_HWPRF	Snoop hit, hardware data prefetch.
0x81A5	DSNP_HIT_NEAR_HWPRF	Snoop hit in near local cache, hardware data prefetch.
0x81A6	DSNP_HIT_FAR_HWPRF	Snoop hit in far local cache, hardware data prefetch.
0x81A7	DSNP_HIT_REMOTE_HWPRF	Snoop hit in remote cache, hardware data prefetch.
0x81A8	ISNP_HIT_PRFM	Snoop hit, software instruction prefetch.
0x81A9	ISNP_HIT_NEAR_PRFM	Snoop hit in near local cache, software instruction prefetch.
0x81AA	ISNP_HIT_FAR_PRFM	Snoop hit in far local cache, software instruction prefetch.
0x81AB	ISNP_HIT_REMOTE_PRFM	Snoop hit in remote cache, software instruction prefetch.
0x81AC	ISNP_HIT_HWPRF	Snoop hit, hardware instruction prefetch.
0x81AD	ISNP_HIT_NEAR_HWPRF	Snoop hit in near local cache, hardware instruction prefetch.
0x81AE	ISNP_HIT_FAR_HWPRF	Snoop hit in far local cache, hardware instruction prefetch.
0x81AF	ISNP_HIT_REMOTE_HWPRF	Snoop hit in remote cache, hardware instruction prefetch.
0x81B0	ISNP_HIT	Snoop hit, instruction.
0x81B1	ISNP_HIT_NEAR	Snoop hit in near local cache, instruction access.
0x81B2	ISNP_HIT_FAR	Snoop hit in far local cache, instruction access.
0x81B3	ISNP_HIT_REMOTE	Snoop hit in remote cache, instruction access.
0x81B4	DSNP_HIT	Snoop hit, data.
0x81B5	DSNP_HIT_NEAR	Snoop hit in near local cache, data access.
0x81B6	DSNP_HIT_FAR	Snoop hit in far local cache, data access.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x81B7	DSNP_HIT_REMOTE	Snoop hit in remote cache, data access.
0x81B8	L1I_CACHE_REFILL_HWPRF	Level 1 instruction cache refill, hardware prefetch.
0x81B9	L2I_CACHE_REFILL_HWPRF	Level 2 instruction cache refill, hardware prefetch.
0x81BC	L1D_CACHE_REFILL_HWPRF	Level 1 data cache refill, hardware prefetch.
0x81BD	L2D_CACHE_REFILL_HWPRF	Level 2 data cache refill, hardware prefetch.
0x81BE	L3D_CACHE_REFILL_HWPRF	Level 3 data cache refill, hardware prefetch.
0x81BF	LL_CACHE_REFILL_HWPRF	Last level cache refill, hardware prefetch.
0x81C0	L1I_CACHE_HIT_RD	Level 1 instruction cache demand fetch hit.
0x81C1	L2I_CACHE_HIT_RD	Level 2 instruction cache demand fetch hit.
0x81C4	L1D_CACHE_HIT_RD	Level 1 data cache demand access hit, read.
0x81C5	L2D_CACHE_HIT_RD	Level 2 data cache demand access hit, read.
0x81C6	L3D_CACHE_HIT_RD	Level 3 data cache demand access hit, read.
0x81C7	LL_CACHE_HIT_RD	Last level cache demand access hit, read.
0x81C8	L1D_CACHE_HIT_WR	Level 1 data cache demand access hit, write.
0x81C9	L2D_CACHE_HIT_WR	Level 2 data cache demand access hit, write.
0x81CA	L3D_CACHE_HIT_WR	Level 3 data cache demand access hit, write.
0x81CB	LL_CACHE_HIT_WR	Last level cache demand access hit, write.
0x81CC	L1D_CACHE_HIT_RW	Level 1 data cache demand access hit.
0x81CD	L2D_CACHE_HIT_RW	Level 2 data cache demand access hit.
0x81CE	L3D_CACHE_HIT_RW	Level 3 data cache demand access hit.
0x81CF	LL_CACHE_HIT_RW	Last level cache demand access hit.
0x81D0	L1I_CACHE_HIT_RD_FPRFM	Level 1 instruction cache demand fetch first hit, fetched by software prefetch.
0x81D1	L2I_CACHE_HIT_RD_FPRFM	Level 2 instruction cache demand fetch first hit, fetched by software prefetch.
0x81D4	L1D_CACHE_HIT_RD_FPRFM	Level 1 data cache demand access first hit, read, fetched by software prefetch.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x81D5	L2D_CACHE_HIT_RD_FPRFM	Level 2 data cache demand access first hit, read, fetched by software prefetch.
0x81D6	L3D_CACHE_HIT_RD_FPRFM	Level 3 data cache demand access first hit, read, fetched by software prefetch.
0x81D7	LL_CACHE_HIT_RD_FPRFM	Last level cache demand access first hit, read, fetched by software prefetch.
0x81D8	L1D_CACHE_HIT_WR_FPRFM	Level 1 data cache demand access first hit, write, fetched by software prefetch.
0x81D9	L2D_CACHE_HIT_WR_FPRFM	Level 2 data cache demand access first hit, write, fetched by software prefetch.
0x81DA	L3D_CACHE_HIT_WR_FPRFM	Level 3 data cache demand access first hit, write, fetched by software prefetch.
0x81DB	LL_CACHE_HIT_WR_FPRFM	Last level cache demand access first hit, write, fetched by software prefetch.
0x81DC	L1D_CACHE_HIT_RW_FPRFM	Level 1 data cache demand access first hit, fetched by software prefetch.
0x81DD	L2D_CACHE_HIT_RW_FPRFM	Level 2 data cache demand access first hit, fetched by software prefetch.
0x81DE	L3D_CACHE_HIT_RW_FPRFM	Level 3 data cache demand access first hit, fetched by software prefetch.
0x81DF	LL_CACHE_HIT_RW_FPRFM	Last level cache demand access first hit, fetched by software prefetch.
0x81E0	L1I_CACHE_HIT_RD_FHWPRF	Level 1 instruction cache demand fetch first hit, fetched by hardware prefetcher.
0x81E1	L2I_CACHE_HIT_RD_FHWPRF	Level 2 instruction cache demand fetch first hit, fetched by hardware prefetcher.
0x81E4	L1D_CACHE_HIT_RD_FHWPRF	Level 1 data cache demand access first hit, read, fetched by hardware prefetcher.
0x81E5	L2D_CACHE_HIT_RD_FHWPRF	Level 2 data cache demand access first hit, read, fetched by hardware prefetcher.
0x81E6	L3D_CACHE_HIT_RD_FHWPRF	Level 3 data cache demand access first hit, read, fetched by hardware prefetcher.
0x81E7	LL_CACHE_HIT_RD_FHWPRF	Last level cache demand access first hit, read, fetched by hardware prefetcher.
0x81E8	L1D_CACHE_HIT_WR_FHWPRF	Level 1 data cache demand access first hit, write, fetched by hardware prefetcher.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x81E9	L2D_CACHE_HIT_WR_FHWPRF	Level 2 data cache demand access first hit, write, fetched by hardware prefetcher.
0x81EA	L3D_CACHE_HIT_WR_FHWPRF	Level 3 data cache demand access first hit, write, fetched by hardware prefetcher.
0x81EB	LL_CACHE_HIT_WR_FHWPRF	Last level cache demand access first hit, write, fetched by hardware prefetcher.
0x81EC	L1D_CACHE_HIT_RW_FHWPRF	Level 1 data cache demand access first hit, fetched by hardware prefetcher.
0x81ED	L2D_CACHE_HIT_RW_FHWPRF	Level 2 data cache demand access first hit, fetched by hardware prefetcher.
0x81EE	L3D_CACHE_HIT_RW_FHWPRF	Level 3 data cache demand access first hit, fetched by hardware prefetcher.
0x81EF	LL_CACHE_HIT_RW_FHWPRF	Last level cache demand access first hit, fetched by hardware prefetcher.
0x81F0	L1I_CACHE_HIT_RD_FPRF	Level 1 instruction cache demand fetch first hit, fetched by prefetch.
0x81F1	L2I_CACHE_HIT_RD_FPRF	Level 2 instruction cache demand fetch first hit, fetched by prefetch.
0x81F4	L1D_CACHE_HIT_RD_FPRF	Level 1 data cache demand access first hit, read, fetched by prefetch.
0x81F5	L2D_CACHE_HIT_RD_FPRF	Level 2 data cache demand access first hit, read, fetched by prefetch.
0x81F6	L3D_CACHE_HIT_RD_FPRF	Level 3 data cache demand access first hit, read, fetched by prefetch.
0x81F7	LL_CACHE_HIT_RD_FPRF	Last level cache demand access first hit, read, fetched by prefetch.
0x81F8	L1D_CACHE_HIT_WR_FPRF	Level 1 data cache demand access first hit, write, fetched by prefetch.
0x81F9	L2D_CACHE_HIT_WR_FPRF	Level 2 data cache demand access first hit, write, fetched by prefetch.
0x81FA	L3D_CACHE_HIT_WR_FPRF	Level 3 data cache demand access first hit, write, fetched by prefetch.
0x81FB	LL_CACHE_HIT_WR_FPRF	Last level cache demand access first hit, write, fetched by prefetch.
0x81FC	L1D_CACHE_HIT_RW_FPRF	Level 1 data cache demand access first hit, fetched by prefetch.
0x81FD	L2D_CACHE_HIT_RW_FPRF	Level 2 data cache demand access first hit, fetched by prefetch.
0x81FE	L3D_CACHE_HIT_RW_FPRF	Level 3 data cache demand access first hit, fetched by prefetch.
0x81FF	LL_CACHE_HIT_RW_FPRF	Last level cache demand access first hit, fetched by prefetch.
0x8200	L1I_CACHE_HIT	Level 1 instruction cache hit.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x8201	L2I_CACHE_HIT	Level 2 instruction cache hit.
0x8204	L1D_CACHE_HIT	Level 1 data cache hit.
0x8205	L2D_CACHE_HIT	Level 2 data cache hit.
0x8206	L3D_CACHE_HIT	Level 3 data cache hit.
0x8207	LL_CACHE_HIT	Last level cache hit.
0x8208	L1I_CACHE_HIT_PRFM	Level 1 instruction cache software prefetch hit.
0x8209	L2I_CACHE_HIT_PRFM	Level 2 instruction cache software prefetch hit.
0x820C	L1D_CACHE_HIT_PRFM	Level 1 data cache software prefetch hit.
0x820D	L2D_CACHE_HIT_PRFM	Level 2 data cache software prefetch hit.
0x820E	L3D_CACHE_HIT_PRFM	Level 3 data cache software prefetch hit.
0x820F	LL_CACHE_HIT_PRFM	Last level cache software prefetch hit.
0x8214	L1D_CACHE_HITM_RD	Level 1 data cache demand access hit modified, read.
0x8215	L2D_CACHE_HITM_RD	Level 2 data cache demand access hit modified, read.
0x8216	L3D_CACHE_HITM_RD	Level 3 data cache demand access hit modified, read.
0x8217	LL_CACHE_HITM_RD	Last level cache demand access hit modified, read.
0x8218	L1D_CACHE_HITM_WR	Level 1 data cache demand access hit modified, write.
0x8219	L2D_CACHE_HITM_WR	Level 2 data cache demand access hit modified, write.
0x821A	L3D_CACHE_HITM_WR	Level 3 data cache demand access hit modified, write.
0x821B	LL_CACHE_HITM_WR	Last level cache demand access hit modified, write.
0x821C	L1D_CACHE_HITM_RW	Level 1 data cache demand access hit modified.
0x821D	L2D_CACHE_HITM_RW	Level 2 data cache demand access hit modified.
0x821E	L3D_CACHE_HITM_RW	Level 3 data cache demand access hit modified.
0x821F	LL_CACHE_HITM_RW	Last level cache demand access hit modified.
0x8224	DSNP_HITM_RD	Snoop hit, demand data read, modified.
0x8225	DSNP_HITM_NEAR_RD	Snoop hit in near local cache, demand data read, modified.
0x8226	DSNP_HITM_FAR_RD	Snoop hit in far local cache, demand data read, modified.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x8227	DSNP_HITM_REMOTE_RD	Snoop hit in remote cache, demand data read, modified.
0x8228	DSNP_HITM_WR	Snoop hit, demand data write, modified.
0x8229	DSNP_HITM_NEAR_WR	Snoop hit in near local cache, demand data write, modified.
0x822A	DSNP_HITM_FAR_WR	Snoop hit in far local cache, demand data write, modified.
0x822B	DSNP_HITM_REMOTE_WR	Snoop hit in remote cache, demand data write, modified.
0x822C	DSNP_HITM_RW	Snoop hit, demand data access, modified.
0x822D	DSNP_HITM_NEAR_RW	Snoop hit in near local cache, demand data access, modified.
0x822E	DSNP_HITM_FAR_RW	Snoop hit in far local cache, demand data access, modified.
0x822F	DSNP_HITM_REMOTE_RW	Snoop hit in remote cache, demand data access, modified.
0x8230	LOCAL_MEM	Access to memory attached to this device.
0x8231	LOCAL_MEM_RD	Access to memory attached to this device, demand access, read.
0x8232	LOCAL_MEM_WR	Access to memory attached to this device, demand access, write.
0x8233	LOCAL_MEM_RW	Access to memory attached to this device, demand access.
0x8234	LOCAL_MEM_PRFM	Access to memory attached to this device, software prefetch.
0x8235	LOCAL_MEM_LD_RETIRED	Load instruction architecturally executed, access to memory attached to this device.
0x8236	LOCAL_MEM_ST_RETIRED	Store instruction architecturally executed, access to memory attached to this device.
0x8237	LOCAL_MEM_LDST_RETIRED	Load or store instruction architecturally executed, access to memory attached to this device.
0x8238	REMOTE_MEM	Access to memory attached to a remote device.
0x8239	REMOTE_MEM_RD	Access to memory attached to a remote device, demand access, read.
0x823A	REMOTE_MEM_WR	Access to memory attached to a remote device, demand access, write.
0x823B	REMOTE_MEM_RW	Access to memory attached to a remote device, demand access.
0x823C	REMOTE_MEM_PRFM	Access to memory attached to a remote device, software prefetch.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x823D	REMOTE_MEM_LD_RETIRED	Load instruction architecturally executed, access to memory attached to a remote device.
0x823E	REMOTE_MEM_ST_RETIRED	Store instruction architecturally executed, access to memory attached to a remote device.
0x823F	REMOTE_MEM_LDST_RETIRED	Load or store instruction architecturally executed, access to memory attached to a remote device.
0x8240	L1I_LFB_HIT_RD	Level 1 instruction cache demand fetch line-fill buffer hit.
0x8241	L2I_LFB_HIT_RD	Level 2 instruction cache demand fetch line-fill buffer hit.
0x8244	L1D_LFB_HIT_RD	Level 1 data cache demand access line-fill buffer hit, read.
0x8245	L2D_LFB_HIT_RD	Level 2 data cache demand access line-fill buffer hit, read.
0x8246	L3D_LFB_HIT_RD	Level 3 data cache demand access line-fill buffer hit, read.
0x8247	LL_LFB_HIT_RD	Last level cache demand access line-fill buffer hit, read.
0x8248	L1D_LFB_HIT_WR	Level 1 data cache demand access line-fill buffer hit, write.
0x8249	L2D_LFB_HIT_WR	Level 2 data cache demand access line-fill buffer hit, write.
0x824A	L3D_LFB_HIT_WR	Level 3 data cache demand access line-fill buffer hit, write.
0x824B	LL_LFB_HIT_WR	Last level cache demand access line-fill buffer hit, write.
0x824C	L1D_LFB_HIT_RW	Level 1 data cache demand access line-fill buffer hit.
0x824D	L2D_LFB_HIT_RW	Level 2 data cache demand access line-fill buffer hit.
0x824E	L3D_LFB_HIT_RW	Level 3 data cache demand access line-fill buffer hit.
0x824F	LL_LFB_HIT_RW	Last level cache demand access line-fill buffer hit.
0x8250	L1I_LFB_HIT_RD_FPRFM	Level 1 instruction cache demand fetch line-fill buffer first hit, recently fetched by software prefetch.
0x8251	L2I_LFB_HIT_RD_FPRFM	Level 2 instruction cache demand fetch line-fill buffer first hit, recently fetched by software prefetch.
0x8254	L1D_LFB_HIT_RD_FPRFM	Level 1 data cache demand access line-fill buffer first hit, read, recently fetched by software prefetch.
0x8255	L2D_LFB_HIT_RD_FPRFM	Level 2 data cache demand access line-fill buffer first hit, read, recently fetched by software prefetch.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x8256	L3D_LFB_HIT_RD_FPRFM	Level 3 data cache demand access line-fill buffer first hit, read, recently fetched by software prefetch.
0x8257	LL_LFB_HIT_RD_FPRFM	Last level cache demand access line-fill buffer first hit, read, recently fetched by software prefetch.
0x8258	L1D_LFB_HIT_WR_FPRFM	Level 1 data cache demand access line-fill buffer first hit, write, recently fetched by software prefetch.
0x8259	L2D_LFB_HIT_WR_FPRFM	Level 2 data cache demand access line-fill buffer first hit, write, recently fetched by software prefetch.
0x825A	L3D_LFB_HIT_WR_FPRFM	Level 3 data cache demand access line-fill buffer first hit, write, recently fetched by software prefetch.
0x825B	LL_LFB_HIT_WR_FPRFM	Last level cache demand access line-fill buffer first hit, write, recently fetched by software prefetch.
0x825C	L1D_LFB_HIT_RW_FPRFM	Level 1 data cache demand access line-fill buffer first hit, recently fetched by software prefetch.
0x825D	L2D_LFB_HIT_RW_FPRFM	Level 2 data cache demand access line-fill buffer first hit, recently fetched by software prefetch.
0x825E	L3D_LFB_HIT_RW_FPRFM	Level 3 data cache demand access line-fill buffer first hit, recently fetched by software prefetch.
0x825F	LL_LFB_HIT_RW_FPRFM	Last level cache demand access line-fill buffer first hit, recently fetched by software prefetch.
0x8260	L1I_LFB_HIT_RD_FHWPRF	Level 1 instruction cache demand fetch line-fill buffer first hit, recently fetched by hardware prefetcher.
0x8261	L2I_LFB_HIT_RD_FHWPRF	Level 2 instruction cache demand fetch line-fill buffer first hit, recently fetched by hardware prefetcher.
0x8264	L1D_LFB_HIT_RD_FHWPRF	Level 1 data cache demand access line-fill buffer first hit, read, recently fetched by hardware prefetcher.
0x8265	L2D_LFB_HIT_RD_FHWPRF	Level 2 data cache demand access line-fill buffer first hit, read, recently fetched by hardware prefetcher.
0x8266	L3D_LFB_HIT_RD_FHWPRF	Level 3 data cache demand access line-fill buffer first hit, read, recently fetched by hardware prefetcher.
0x8267	LL_LFB_HIT_RD_FHWPRF	Last level cache demand access line-fill buffer first hit, read, recently fetched by hardware prefetcher.
0x8268	L1D_LFB_HIT_WR_FHWPRF	Level 1 data cache demand access line-fill buffer first hit, write, recently fetched by hardware prefetcher.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x8269	L2D_LFB_HIT_WR_FHWPRF	Level 2 data cache demand access line-fill buffer first hit, write, recently fetched by hardware prefetcher.
0x826A	L3D_LFB_HIT_WR_FHWPRF	Level 3 data cache demand access line-fill buffer first hit, write, recently fetched by hardware prefetcher.
0x826B	LL_LFB_HIT_WR_FHWPRF	Last level cache demand access line-fill buffer first hit, write, recently fetched by hardware prefetcher.
0x826C	L1D_LFB_HIT_RW_FHWPRF	Level 1 data cache demand access line-fill buffer first hit, recently fetched by hardware prefetcher.
0x826D	L2D_LFB_HIT_RW_FHWPRF	Level 2 data cache demand access line-fill buffer first hit, recently fetched by hardware prefetcher.
0x826E	L3D_LFB_HIT_RW_FHWPRF	Level 3 data cache demand access line-fill buffer first hit, recently fetched by hardware prefetcher.
0x826F	LL_LFB_HIT_RW_FHWPRF	Last level cache demand access line-fill buffer first hit, recently fetched by hardware prefetcher.
0x8270	L1I_LFB_HIT_RD_FPRF	Level 1 instruction cache demand fetch line-fill buffer first hit, recently fetched by prefetch.
0x8271	L2I_LFB_HIT_RD_FPRF	Level 2 instruction cache demand fetch line-fill buffer first hit, recently fetched by prefetch.
0x8274	L1D_LFB_HIT_RD_FPRF	Level 1 data cache demand access line-fill buffer first hit, read, recently fetched by prefetch.
0x8275	L2D_LFB_HIT_RD_FPRF	Level 2 data cache demand access line-fill buffer first hit, read, recently fetched by prefetch.
0x8276	L3D_LFB_HIT_RD_FPRF	Level 3 data cache demand access line-fill buffer first hit, read, recently fetched by prefetch.
0x8277	LL_LFB_HIT_RD_FPRF	Last level cache demand access line-fill buffer first hit, read, recently fetched by prefetch.
0x8278	L1D_LFB_HIT_WR_FPRF	Level 1 data cache demand access line-fill buffer first hit, write, recently fetched by prefetch.
0x8279	L2D_LFB_HIT_WR_FPRF	Level 2 data cache demand access line-fill buffer first hit, write, recently fetched by prefetch.
0x827A	L3D_LFB_HIT_WR_FPRF	Level 3 data cache demand access line-fill buffer first hit, write, recently fetched by prefetch.
0x827B	LL_LFB_HIT_WR_FPRF	Last level cache demand access line-fill buffer first hit, write, recently fetched by prefetch.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x827C	L1D_LFB_HIT_RW_FPRF	Level 1 data cache demand access line-fill buffer first hit, recently fetched by prefetch.
0x827D	L2D_LFB_HIT_RW_FPRF	Level 2 data cache demand access line-fill buffer first hit, recently fetched by prefetch.
0x827E	L3D_LFB_HIT_RW_FPRF	Level 3 data cache demand access line-fill buffer first hit, recently fetched by prefetch.
0x827F	LL_LFB_HIT_RW_FPRF	Last level cache demand access line-fill buffer first hit, recently fetched by prefetch.
0x8280	L1I_CACHE_PRF	Level 1 instruction cache, prefetch hit.
0x8281	L2I_CACHE_PRF	Level 2 instruction cache, prefetch hit.
0x8284	L1D_CACHE_PRF	Level 1 data cache, prefetch hit.
0x8285	L2D_CACHE_PRF	Level 2 data cache, prefetch hit.
0x8286	L3D_CACHE_PRF	Level 3 data cache, prefetch hit.
0x8287	LL_CACHE_PRF	Last level cache, prefetch hit.
0x8288	L1I_CACHE_REFILL_PRF	Level 1 instruction cache refill, prefetch hit.
0x8289	L2I_CACHE_REFILL_PRF	Level 2 instruction cache refill, prefetch hit.
0x828C	L1D_CACHE_REFILL_PRF	Level 1 data cache refill, prefetch hit.
0x828D	L2D_CACHE_REFILL_PRF	Level 2 data cache refill, prefetch hit.
0x828E	L3D_CACHE_REFILL_PRF	Level 3 data cache refill, prefetch hit.
0x828F	LL_CACHE_REFILL_PRF	Last level cache refill, prefetch hit.
0x8290	ISNP_HIT_PRF	Snoop hit, instruction prefetch.
0x8291	ISNP_HIT_NEAR_PRF	Snoop hit in near local cache, instruction prefetch.
0x8292	ISNP_HIT_FAR_PRF	Snoop hit in far local cache, instruction prefetch.
0x8293	ISNP_HIT_REMOTE_PRF	Snoop hit in remote cache, instruction prefetch.
0x8294	DSNP_HIT_PRF	Snoop hit, data prefetch.
0x8295	DSNP_HIT_NEAR_PRF	Snoop hit in near local cache, data prefetch.
0x8296	DSNP_HIT_FAR_PRF	Snoop hit in far local cache, data prefetch.
0x8297	DSNP_HIT_REMOTE_PRF	Snoop hit in remote cache, data prefetch.
0x8298	LL_CACHE_RW	Last level cache demand access.
0x8299	LL_CACHE_PRFM	Last level cache software prefetch.
0x829A	LL_CACHE_REFILL	Last level cache refill.
0x829B	LL_CACHE_REFILL_PRFM	Last level cache refill, software prefetch.
0x829C	LL_CACHE_WB	Last level cache write-back.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x829D	LL_CACHE_WR	Last level cache access, write.
0x829F	LL_CACHE_REFILL_WR	Last level cache refill, write.
0x82A0	MEM_ACCESS_RW	Data memory access, demand access.
0x82A1	INST_FETCH_RD	Instruction memory access, demand fetch.
0x82A2	MEM_ACCESS_PRFM	Data memory access, software prefetch.
0x82A3	INST_FETCH_PRFM	Instruction memory access, software prefetch.
0x82A4	ASE_SVE_RETIRED	Instruction architecturally executed, Advanced SIMD data processing or SVE data processing.
0x82A8	LD_ANY_RETIRED	Instruction architecturally executed, load.
0x82A9	ST_ANY_RETIRED	Instruction architecturally executed, store.
0x82AA	LDST_ANY_RETIRED	Instruction architecturally executed, load or store.
0x82AB	DP_RETIRED	Instruction architecturally executed, integer data processing.
0x82AC	ASE_RETIRED	Instruction architecturally executed, Advanced SIMD data processing.
0x82AD	VFP_RETIRED	Instruction architecturally executed, scalar floating-point data processing.
0x82AF	CRYPTO_RETIRED	Instruction architecturally executed, cryptographic data processing.
0x82B0	L1I_CACHE_MISS_RETIRED	Instruction architecturally executed, miss in Level 1 instruction cache.
0x82B1	L2I_CACHE_MISS_RETIRED	Instruction architecturally executed, miss in Level 2 instruction cache.
0x82B3	PRF_RETIRED	Instruction architecturally executed, prefetch.
0x82B4	L1D_CACHE_MISS_LD_RETIRED	Load instruction architecturally executed, miss in Level 1 data cache.
0x82B5	L2D_CACHE_MISS_LD_RETIRED	Load instruction architecturally executed, miss in Level 2 data cache.
0x82B6	L3D_CACHE_MISS_LD_RETIRED	Load instruction architecturally executed, miss in Level 3 data cache.
0x82B7	LL_CACHE_MISS_LD_RETIRED	Load instruction architecturally executed, miss in Last level cache.
0x82B8	L1D_CACHE_MISS_ST_RETIRED	Store instruction architecturally executed, miss in Level 1 data cache.
0x82B9	L2D_CACHE_MISS_ST_RETIRED	Store instruction architecturally executed, miss in Level 2 data cache.
0x82BA	L3D_CACHE_MISS_ST_RETIRED	Store instruction architecturally executed, miss in Level 3 data cache.
0x82BB	LL_CACHE_MISS_ST_RETIRED	Store instruction architecturally executed, miss in Last level cache.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x82BC	L1D_CACHE_MISS_LDST_RETIRED	Load or store instruction architecturally executed, miss in Level 1 data cache.
0x82BD	L2D_CACHE_MISS_LDST_RETIRED	Load or store instruction architecturally executed, miss in Level 2 data cache.
0x82BE	L3D_CACHE_MISS_LDST_RETIRED	Load or store instruction architecturally executed, miss in Level 3 data cache.
0x82BF	LL_CACHE_MISS_LDST_RETIRED	Load or store instruction architecturally executed, miss in Last level cache.
0x82C4	L1D_CACHE_HITM_LD_RETIRED	Load instruction architecturally executed, hit modified data in Level 1 data cache.
0x82C5	L2D_CACHE_HITM_LD_RETIRED	Load instruction architecturally executed, hit modified data in Level 2 data cache.
0x82C6	L3D_CACHE_HITM_LD_RETIRED	Load instruction architecturally executed, hit modified data in Level 3 data cache.
0x82C7	LL_CACHE_HITM_LD_RETIRED	Load instruction architecturally executed, hit modified data in Last level cache.
0x82C8	L1D_CACHE_HITM_ST_RETIRED	Store instruction architecturally executed, hit modified data in Level 1 data cache.
0x82C9	L2D_CACHE_HITM_ST_RETIRED	Store instruction architecturally executed, hit modified data in Level 2 data cache.
0x82CA	L3D_CACHE_HITM_ST_RETIRED	Store instruction architecturally executed, hit modified data in Level 3 data cache.
0x82CB	LL_CACHE_HITM_ST_RETIRED	Store instruction architecturally executed, hit modified data in Last level cache.
0x82CC	L1D_CACHE_HITM_LDST_RETIRED	Load or store instruction architecturally executed, hit modified data in Level 1 data cache.
0x82CD	L2D_CACHE_HITM_LDST_RETIRED	Load or store instruction architecturally executed, hit modified data in Level 2 data cache.
0x82CE	L3D_CACHE_HITM_LDST_RETIRED	Load or store instruction architecturally executed, hit modified data in Level 3 data cache.
0x82CF	LL_CACHE_HITM_LDST_RETIRED	Load or store instruction architecturally executed, hit modified data in Last level cache.
0x82D0	L1I_LFB_HIT_RETIRED	Instruction architecturally executed, line-fill buffer hit in Level 1 instruction cache.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x82D1	L2I_LFB_HIT_RETIRED	Instruction architecturally executed, line-fill buffer hit in Level 2 instruction cache.
0x82D4	L1D_LFB_HIT_LD_RETIRED	Load instruction architecturally executed, line-fill buffer hit in Level 1 data cache.
0x82D5	L2D_LFB_HIT_LD_RETIRED	Load instruction architecturally executed, line-fill buffer hit in Level 2 data cache.
0x82D6	L3D_LFB_HIT_LD_RETIRED	Load instruction architecturally executed, line-fill buffer hit in Level 3 data cache.
0x82D7	LL_LFB_HIT_LD_RETIRED	Load instruction architecturally executed, line-fill buffer hit in Last level cache.
0x82D8	L1D_LFB_HIT_ST_RETIRED	Store instruction architecturally executed, line-fill buffer hit in Level 1 data cache.
0x82D9	L2D_LFB_HIT_ST_RETIRED	Store instruction architecturally executed, line-fill buffer hit in Level 2 data cache.
0x82DA	L3D_LFB_HIT_ST_RETIRED	Store instruction architecturally executed, line-fill buffer hit in Level 3 data cache.
0x82DB	LL_LFB_HIT_ST_RETIRED	Store instruction architecturally executed, line-fill buffer hit in Last level cache.
0x82DC	L1D_LFB_HIT_LDST_RETIRED	Load or store instruction architecturally executed, line-fill buffer hit in Level 1 data cache.
0x82DD	L2D_LFB_HIT_LDST_RETIRED	Load or store instruction architecturally executed, line-fill buffer hit in Level 2 data cache.
0x82DE	L3D_LFB_HIT_LDST_RETIRED	Load or store instruction architecturally executed, line-fill buffer hit in Level 3 data cache.
0x82DF	LL_LFB_HIT_LDST_RETIRED	Load or store instruction architecturally executed, line-fill buffer hit in Last level cache.
0x82E0	L1I_CACHE_HIT_RETIRED	Instruction architecturally executed, hit in Level 1 instruction cache.
0x82E1	L2I_CACHE_HIT_RETIRED	Instruction architecturally executed, hit in Level 2 instruction cache.
0x82E4	L1D_CACHE_HIT_LD_RETIRED	Load instruction architecturally executed, hit in Level 1 data cache.
0x82E5	L2D_CACHE_HIT_LD_RETIRED	Load instruction architecturally executed, hit in Level 2 data cache.
0x82E6	L3D_CACHE_HIT_LD_RETIRED	Load instruction architecturally executed, hit in Level 3 data cache.
0x82E7	LL_CACHE_HIT_LD_RETIRED	Load instruction architecturally executed, hit in Last level cache.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x82E8	L1D_CACHE_HIT_ST_RETIRED	Store instruction architecturally executed, hit in Level 1 data cache.
0x82E9	L2D_CACHE_HIT_ST_RETIRED	Store instruction architecturally executed, hit in Level 2 data cache.
0x82EA	L3D_CACHE_HIT_ST_RETIRED	Store instruction architecturally executed, hit in Level 3 data cache.
0x82EB	LL_CACHE_HIT_ST_RETIRED	Store instruction architecturally executed, hit in Last level cache.
0x82EC	L1D_CACHE_HIT_LDST_RETIRED	Load or store instruction architecturally executed, hit in Level 1 data cache.
0x82ED	L2D_CACHE_HIT_LDST_RETIRED	Load or store instruction architecturally executed, hit in Level 2 data cache.
0x82EE	L3D_CACHE_HIT_LDST_RETIRED	Load or store instruction architecturally executed, hit in Level 3 data cache.
0x82EF	LL_CACHE_HIT_LDST_RETIRED	Load or store instruction architecturally executed, hit in Last level cache.
0x82F0	ITLB_HIT_RETIRED	Instruction architecturally executed, no translation table walk.
0x82F1	DTLB_HIT_LD_RETIRED	Load instruction architecturally executed, no translation table walk.
0x82F2	DTLB_HIT_ST_RETIRED	Store instruction architecturally executed, no translation table walk.
0x82F3	DTLB_HIT_LDST_RETIRED	Load or store instruction architecturally executed, no translation table walk.
0x82F4	ITLB_WALK_RETIRED	Instruction architecturally executed, at least one translation table walk.
0x82F5	DTLB_WALK_LD_RETIRED	Load instruction architecturally executed, at least one translation table walk.
0x82F6	DTLB_WALK_ST_RETIRED	Store instruction architecturally executed, at least one translation table walk.
0x82F7	DTLB_WALK_LDST_RETIRED	Load or store instruction architecturally executed, at least one translation table walk.
0x82F8	DTLB_WALK_PRF	Data TLB prefetch, with at least one translation table walk.
0x82F9	ITLB_WALK_PRF	Instruction TLB prefetch, with at least one translation table walk.
0x82FA	DTLB_WALK_HWPRF	Data TLB hardware prefetch, with at least one translation table walk.
0x82FB	ITLB_WALK_HWPRF	Instruction TLB hardware prefetch, with at least one translation table walk.
0x82FC	L1D_TLB_PRF	Level 1 data TLB access, prefetch.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x82FD	L1I_TLB_PRF	Level 1 instruction TLB access, prefetch.
0x82FE	L1D_TLB_HWPRF	Level 1 data TLB access, hardware prefetch.
0x82FF	L1I_TLB_HWPRF	Level 1 instruction TLB access, hardware prefetch.
0x8304	DSNP_HITM_LD_RETIRED	Load instruction architecturally executed, snoop hit.
0x8305	DSNP_HITM_NEAR_LD_RETIRED	Load instruction architecturally executed, snoop hit in near cache.
0x8306	DSNP_HITM_FAR_LD_RETIRED	Load instruction architecturally executed, snoop hit in far cache.
0x8307	DSNP_HITM_REMOTE_LD_RETIRED	Load instruction architecturally executed, snoop hit in remote cache.
0x8308	DSNP_HITM_ST_RETIRED	Store instruction architecturally executed, snoop hit.
0x8309	DSNP_HITM_NEAR_ST_RETIRED	Store instruction architecturally executed, snoop hit in near cache.
0x830A	DSNP_HITM_FAR_ST_RETIRED	Store instruction architecturally executed, snoop hit in far cache.
0x830B	DSNP_HITM_REMOTE_ST_RETIRED	Store instruction architecturally executed, snoop hit in remote cache.
0x830C	DSNP_HITM_LDST_RETIRED	Load or store instruction architecturally executed, snoop hit.
0x830D	DSNP_HITM_NEAR_LDST_RETIRED	Load or store instruction architecturally executed, snoop hit in near cache.
0x830E	DSNP_HITM_FAR_LDST_RETIRED	Load or store instruction architecturally executed, snoop hit in far cache.
0x830F	DSNP_HITM_REMOTE_LDST_RETIRED	Load or store instruction architecturally executed, snoop hit in remote cache.
0x8310	ISNP_HIT_RETIRED	Instruction architecturally executed, instruction fetch snoop hit.
0x8311	ISNP_HIT_NEAR_RETIRED	Instruction architecturally executed, instruction fetch snoop hit in near cache.
0x8312	ISNP_HIT_FAR_RETIRED	Instruction architecturally executed, instruction fetch snoop hit in far cache.
0x8313	ISNP_HIT_REMOTE_RETIRED	Instruction architecturally executed, instruction fetch snoop hit in remote cache.
0x8314	DSNP_HIT_LD_RETIRED	Load instruction architecturally executed, snoop hit.
0x8315	DSNP_HIT_NEAR_LD_RETIRED	Load instruction architecturally executed, snoop hit in near cache.
0x8316	DSNP_HIT_FAR_LD_RETIRED	Load instruction architecturally executed, snoop hit in far cache.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x8317	DSNP_HIT_REMOTE_LD_RETIRED	Load instruction architecturally executed, snoop hit in remote cache.
0x8318	DSNP_HIT_ST_RETIRED	Store instruction architecturally executed, snoop hit.
0x8319	DSNP_HIT_NEAR_ST_RETIRED	Store instruction architecturally executed, snoop hit in near cache.
0x831A	DSNP_HIT_FAR_ST_RETIRED	Store instruction architecturally executed, snoop hit in far cache.
0x831B	DSNP_HIT_REMOTE_ST_RETIRED	Store instruction architecturally executed, snoop hit in remote cache.
0x831C	DSNP_HIT_LDST_RETIRED	Load or store instruction architecturally executed, snoop hit.
0x831D	DSNP_HIT_NEAR_LDST_RETIRED	Load or store instruction architecturally executed, snoop hit in near cache.
0x831E	DSNP_HIT_FAR_LDST_RETIRED	Load or store instruction architecturally executed, snoop hit in far cache.
0x831F	DSNP_HIT_REMOTE_LDST_RETIRED	Load or store instruction architecturally executed, snoop hit in remote cache.
0x8320	L1D_CACHE_REFILL_PERCYC	Level 1 data or unified cache refills in progress.
0x8321	L2D_CACHE_REFILL_PERCYC	Level 2 data or unified cache refills in progress.
0x8322	L3D_CACHE_REFILL_PERCYC	Level 3 data or unified cache refills in progress.
0x8324	L1I_CACHE_REFILL_PERCYC	Level 1 instruction or unified cache refills in progress.
0x8325	L2I_CACHE_REFILL_PERCYC	Level 2 instruction or unified cache refills in progress.
0x8328	REMOTE_CACHE_HIT_RD	Remote cache demand access hit, read.
0x8329	REMOTE_CACHE_HITM_RD	Remote cache demand access hit modified, read.
0x832A	REMOTE_LFB_HIT_RD	Remote cache demand access line-fill buffer hit, read.
0x832B	REMOTE_CACHE_HIT_PRFM	Remote cache software prefetch hit.
0x8330	L1GCS_CACHE	Level 1 GCS cache access.
0x8331	L1GCS_CACHE_RW	Level 1 GCS cache demand access.
0x8332	L1GCS_CACHE_HWPRF	Level 1 GCS cache hardware prefetch.
0x8334	L1GCS_CACHE_MISS	Level 1 GCS cache demand access miss.
0x8335	L1GCS_CACHE_MISS_RD	Level 1 GCS cache demand access read miss.
0x8336	L1GCS_CACHE_HIT_RW	Level 1 GCS cache demand access hit.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x8337	L1GCS_CACHE_HIT_RW_FHWPRF	Level 1 GCS cache demand access first hit, fetched by GCS hardware prefetcher.
0x8338	L1GCS_CACHE_REFILL	Level 1 GCS cache refill.
0x8339	L1GCS_CACHE_REFILL_HWPRF	Level 1 GCS cache refill, hardware prefetch.
0x833A	L1GCS_CACHE_REFILL_PERCYC	Level 1 GCS cache refills in progress.
0x833C	L1GCS_LFB_HIT_RW	Level 1 GCS cache demand access line-fill buffer hit.
0x833D	L1GCS_LFB_HIT_RW_FHWPRF	Level 1 GCS cache demand access line-fill buffer first hit, recently fetched by GCS hardware prefetcher.
0x833E	L1GCS_CACHE_INVAL	Level 1 GCS cache invalidate.
0x8340	L1GCS_TLB	Level 1 GCS TLB access.
0x8341	L1GCS_TLB_RW	Level 1 GCS demand TLB access.
0x8342	L1GCS_TLB_HWPRF	Level 1 GCS demand TLB access, GCS hardware prefetch.
0x8344	GCSTLB_WALK	GCS TLB access with at least one translation table walk.
0x8345	GCSTLB_WALK_RW	GCS TLB demand access with at least one translation table walk.
0x8346	GCSTLB_WALK_PERCYC	Translation table walks in progress.
0x8347	GCSTLB_WALK_STEP	GCS TLB translation table walk, step.
0x8350	STALL_BACKEND_L1GCS	Backend stall cycles, Level 1 GCS cache.
0x8351	STALL_BACKEND_GCSTLB	Backend stall cycles, GCS TLB.
0x8353	SE_FP_SPEC	Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD floating-point, data-processing.
0x8359	SE_RETIRED	Instruction architecturally executed, Advanced SIMD, SVE, or SME data processing.
0x835B	SE_INST_RETIRED	Instruction architecturally executed, Advanced SIMD, SVE, or SME.
0x835D	SE_SPEC	Operation speculatively executed, Advanced SIMD, SVE, or SME data processing.
0x835F	SE_INST_SPEC	Operation speculatively executed, Advanced SIMD, SVE, or SME.
0x8361	SE_INT8_SPEC	Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD 8-bit integer, data-processing.
0x8363	SE_FP_BF16_SPEC	Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD BFloat16 floating-point, data-processing.
0x8365	SE_INT16_SPEC	Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD 16-bit integer, data-processing.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x8367	SE_FP_HP_SPEC	Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD half-precision floating-point, data-processing.
0x8369	SE_INT32_SPEC	Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD 32-bit integer, data-processing.
0x836B	SE_FP_SP_SPEC	Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD single-precision floating-point, data-processing.
0x836D	SE_INT64_SPEC	Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD 64-bit integer, data-processing.
0x836F	SE_FP_DP_SPEC	Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD double-precision floating-point, data-processing.
0x8371	SE_FP_ADDSUB_SPEC	Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD floating-point addition or subtraction.
0x8373	SE_FP_FMA_SPEC	Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD floating-point multiply-add or multiply-subtract.
0x8375	SE_FP_DOT_SPEC	Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD floating-point dot product.
0x8379	SE_INT_SPEC	Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD integer, data-processing.
0x837B	SE_INT_MUL_SPEC	Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD integer multiply or multiply-accumulate.
0x837D	SE_INT_DOT_SPEC	Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD integer dot product.
0x8380	ZA_ACTIVE	CPU cycles, ZA enabled.
0x8398	ASE_LUT_SPEC	Lookup table operation speculatively executed, Advanced SIMD data processing.
0x8399	SVE_LUT_SPEC	Lookup table operation speculatively executed, SVE data processing.
0x839B	SE_LUT_SPEC	Lookup table operation speculatively executed, Advanced SIMD, SVE, or SME data processing.
0x83A0	N1_CACHE_HIT_RD	Cache at distance 1, demand access hit, read.
0x83A1	N2_CACHE_HIT_RD	Cache at distance 2, demand access hit, read.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x83A2	N3_CACHE_HIT_RD	Cache at distance 3, demand access hit, read.
0x83A3	N4_CACHE_HIT_RD	Cache at distance 4, demand access hit, read.
0x83A4	N1_CACHE_HIT_PRFM	Cache at distance 1, software prefetch hit.
0x83A5	N2_CACHE_HIT_PRFM	Cache at distance 2, software prefetch hit.
0x83A6	N3_CACHE_HIT_PRFM	Cache at distance 3, software prefetch hit.
0x83A7	N4_CACHE_HIT_PRFM	Cache at distance 4, software prefetch hit.
0x83A8	N1_CACHE1_HIT_RD	Cache type 1 at distance 1 demand access hit, read.
0x83A9	N2_CACHE1_HIT_RD	Cache type 1 at distance 2 demand access hit, read.
0x83AA	N3_CACHE1_HIT_RD	Cache type 1 at distance 3 demand access hit, read.
0x83AB	N4_CACHE1_HIT_RD	Cache type 1 at distance 4 demand access hit, read.
0x83AC	N1_CACHE1_HIT_PRFM	Cache type 1 at distance 1 software prefetch hit.
0x83AD	N2_CACHE1_HIT_PRFM	Cache type 1 at distance 2 software prefetch hit.
0x83AE	N3_CACHE1_HIT_PRFM	Cache type 1 at distance 3 software prefetch hit.
0x83AF	N4_CACHE1_HIT_PRFM	Cache type 1 at distance 4 software prefetch hit.
0x83B0	N1_CACHE2_HIT_RD	Cache type 2 at distance 1 demand access hit, read.
0x83B1	N2_CACHE2_HIT_RD	Cache type 2 at distance 2 demand access hit, read.
0x83B2	N3_CACHE2_HIT_RD	Cache type 2 at distance 3 demand access hit, read.
0x83B3	N4_CACHE2_HIT_RD	Cache type 2 at distance 4 demand access hit, read.
0x83B4	N1_CACHE2_HIT_PRFM	Cache type 2 at distance 1 software prefetch hit.
0x83B5	N2_CACHE2_HIT_PRFM	Cache type 2 at distance 2 software prefetch hit.
0x83B6	N3_CACHE2_HIT_PRFM	Cache type 2 at distance 3 software prefetch hit.
0x83B7	N4_CACHE2_HIT_PRFM	Cache type 2 at distance 4 software prefetch hit.
0x83B8	N1_CACHE3_HIT_RD	Cache type 3 at distance 1 demand access hit, read.
0x83B9	N2_CACHE3_HIT_RD	Cache type 3 at distance 2 demand access hit, read.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x83BA	N3_CACHE3_HIT_RD	Cache type 3 at distance 3 demand access hit, read.
0x83BB	N4_CACHE3_HIT_RD	Cache type 3 at distance 4 demand access hit, read.
0x83BC	N1_CACHE3_HIT_PRFM	Cache type 3 at distance 1 software prefetch hit.
0x83BD	N2_CACHE3_HIT_PRFM	Cache type 3 at distance 2 software prefetch hit.
0x83BE	N3_CACHE3_HIT_PRFM	Cache type 3 at distance 3 software prefetch hit.
0x83BF	N4_CACHE3_HIT_PRFM	Cache type 3 at distance 4 software prefetch hit.
0x83C0	N1_CACHE_HITM_RD	Cache at distance 1, demand access hit modified, read.
0x83C1	N2_CACHE_HITM_RD	Cache at distance 2, demand access hit modified, read.
0x83C2	N3_CACHE_HITM_RD	Cache at distance 3, demand access hit modified, read.
0x83C3	N4_CACHE_HITM_RD	Cache at distance 4, demand access hit modified, read.
0x83C4	N1_LFB_HIT_RD	Cache at distance 1 demand access line-fill buffer hit, read.
0x83C5	N2_LFB_HIT_RD	Cache at distance 2 demand access line-fill buffer hit, read.
0x83C6	N3_LFB_HIT_RD	Cache at distance 3 demand access line-fill buffer hit, read.
0x83C7	N4_LFB_HIT_RD	Cache at distance 4 demand access line-fill buffer hit, read.
0x83C8	N1_CACHE1_HITM_RD	Cache type 1 at distance 1 demand access hit modified, read.
0x83C9	N2_CACHE1_HITM_RD	Cache type 1 at distance 2 demand access hit modified, read.
0x83CA	N3_CACHE1_HITM_RD	Cache type 1 at distance 3 demand access hit modified, read.
0x83CB	N4_CACHE1_HITM_RD	Cache type 1 at distance 4 demand access hit modified, read.
0x83CC	N1_LFB1_HIT_RD	Cache type 1 at distance 1 demand access line-fill buffer hit, read.
0x83CD	N2_LFB1_HIT_RD	Cache type 1 at distance 2 demand access line-fill buffer hit, read.
0x83CE	N3_LFB1_HIT_RD	Cache type 1 at distance 3 demand access line-fill buffer hit, read.
0x83CF	N4_LFB1_HIT_RD	Cache type 1 at distance 4 demand access line-fill buffer hit, read.
0x83D0	N1_CACHE2_HITM_RD	Cache type 2 at distance 1 demand access hit modified, read.
0x83D1	N2_CACHE2_HITM_RD	Cache type 2 at distance 2 demand access hit modified, read.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x83D2	N3_CACHE2_HITM_RD	Cache type 2 at distance 3 demand access hit modified, read.
0x83D3	N4_CACHE2_HITM_RD	Cache type 2 at distance 4 demand access hit modified, read.
0x83D4	N1_LFB2_HIT_RD	Cache type 2 at distance 1 demand access line-fill buffer hit, read.
0x83D5	N2_LFB2_HIT_RD	Cache type 2 at distance 2 demand access line-fill buffer hit, read.
0x83D6	N3_LFB2_HIT_RD	Cache type 2 at distance 3 demand access line-fill buffer hit, read.
0x83D7	N4_LFB2_HIT_RD	Cache type 2 at distance 4 demand access line-fill buffer hit, read.
0x83D8	N1_CACHE3_HITM_RD	Cache type 3 at distance 1 demand access hit modified, read.
0x83D9	N2_CACHE3_HITM_RD	Cache type 3 at distance 2 demand access hit modified, read.
0x83DA	N3_CACHE3_HITM_RD	Cache type 3 at distance 3 demand access hit modified, read.
0x83DB	N4_CACHE3_HITM_RD	Cache type 3 at distance 4 demand access hit modified, read.
0x83DC	N1_LFB3_HIT_RD	Cache type 3 at distance 1 demand access line-fill buffer hit, read.
0x83DD	N2_LFB3_HIT_RD	Cache type 3 at distance 2 demand access line-fill buffer hit, read.
0x83DE	N3_LFB3_HIT_RD	Cache type 3 at distance 3 demand access line-fill buffer hit, read.
0x83DF	N4_LFB3_HIT_RD	Cache type 3 at distance 4 demand access line-fill buffer hit, read.
0x83E0	N1_MEM_RD	Access to memory at distance 1, demand access, read.
0x83E1	N2_MEM_RD	Access to memory at distance 2, demand access, read.
0x83E2	N3_MEM_RD	Access to memory at distance 3, demand access, read.
0x83E3	N4_MEM_RD	Access to memory at distance 4, demand access, read.
0x83E4	N1_MEM_PRFM	Access to memory at distance 1, software prefetch.
0x83E5	N2_MEM_PRFM	Access to memory at distance 2, software prefetch.
0x83E6	N3_MEM_PRFM	Access to memory at distance 3, software prefetch.
0x83E7	N4_MEM_PRFM	Access to memory at distance 4, software prefetch.
0x83E8	N1_MEM1_RD	Access to type 1 memory at distance 1, demand access, read.
0x83E9	N2_MEM1_RD	Access to type 1 memory at distance 2, demand access, read.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x83EA	N3_MEM1_RD	Access to type 1 memory at distance 3, demand access, read.
0x83EB	N4_MEM1_RD	Access to type 1 memory at distance 4, demand access, read.
0x83EC	N1_MEM1_PRFM	Access to type 1 memory at distance 1, software prefetch.
0x83ED	N2_MEM1_PRFM	Access to type 1 memory at distance 2, software prefetch.
0x83EE	N3_MEM1_PRFM	Access to type 1 memory at distance 3, software prefetch.
0x83EF	N4_MEM1_PRFM	Access to type 1 memory at distance 4, software prefetch.
0x83F0	N1_MEM2_RD	Access to type 2 memory at distance 1, demand access, read.
0x83F1	N2_MEM2_RD	Access to type 2 memory at distance 2, demand access, read.
0x83F2	N3_MEM2_RD	Access to type 2 memory at distance 3, demand access, read.
0x83F3	N4_MEM2_RD	Access to type 2 memory at distance 4, demand access, read.
0x83F4	N1_MEM2_PRFM	Access to type 2 memory at distance 1, software prefetch.
0x83F5	N2_MEM2_PRFM	Access to type 2 memory at distance 2, software prefetch.
0x83F6	N3_MEM2_PRFM	Access to type 2 memory at distance 3, software prefetch.
0x83F7	N4_MEM2_PRFM	Access to type 2 memory at distance 4, software prefetch.
0x83F8	N1_MEM3_RD	Access to type 3 memory at distance 1, demand access, read.
0x83F9	N2_MEM3_RD	Access to type 3 memory at distance 2, demand access, read.
0x83FA	N3_MEM3_RD	Access to type 3 memory at distance 3, demand access, read.
0x83FB	N4_MEM3_RD	Access to type 3 memory at distance 4, demand access, read.
0x83FC	N1_MEM3_PRFM	Access to type 3 memory at distance 1, software prefetch.
0x83FD	N2_MEM3_PRFM	Access to type 3 memory at distance 2, software prefetch.
0x83FE	N3_MEM3_PRFM	Access to type 3 memory at distance 3, software prefetch.
0x83FF	N4_MEM3_PRFM	Access to type 3 memory at distance 4, software prefetch.
0x8400	ISNP_HIT_N1_RD	Snoop hit in cache at distance 1, demand instruction fetch.
0x8401	ISNP_HIT_N2_RD	Snoop hit in cache at distance 2, demand instruction fetch.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x8402	ISNP_HIT_N3_RD	Snoop hit in cache at distance 3, demand instruction fetch.
0x8403	ISNP_HIT_N4_RD	Snoop hit in cache at distance 4, demand instruction fetch.
0x8404	DSNP_HIT_N1_RD	Snoop hit in cache at distance 1, demand data read.
0x8405	DSNP_HIT_N2_RD	Snoop hit in cache at distance 2, demand data read.
0x8406	DSNP_HIT_N3_RD	Snoop hit in cache at distance 3, demand data read.
0x8407	DSNP_HIT_N4_RD	Snoop hit in cache at distance 4, demand data read.
0x8408	DSNP_HIT_N1_WR	Snoop hit in cache at distance 1, demand data write.
0x8409	DSNP_HIT_N2_WR	Snoop hit in cache at distance 2, demand data write.
0x840A	DSNP_HIT_N3_WR	Snoop hit in cache at distance 3, demand data write.
0x840B	DSNP_HIT_N4_WR	Snoop hit in cache at distance 4, demand data write.
0x840C	DSNP_HIT_N1_RW	Snoop hit in cache at distance 1, demand data access.
0x840D	DSNP_HIT_N2_RW	Snoop hit in cache at distance 2, demand data access.
0x840E	DSNP_HIT_N3_RW	Snoop hit in cache at distance 3, demand data access.
0x840F	DSNP_HIT_N4_RW	Snoop hit in cache at distance 4, demand data access.
0x8410	DSNP_HIT_N1_PRFM	Snoop hit in cache at distance 1, software data prefetch.
0x8411	DSNP_HIT_N2_PRFM	Snoop hit in cache at distance 2, software data prefetch.
0x8412	DSNP_HIT_N3_PRFM	Snoop hit in cache at distance 3, software data prefetch.
0x8413	DSNP_HIT_N4_PRFM	Snoop hit in cache at distance 4, software data prefetch.
0x8414	DSNP_HIT_N1_HWPRF	Snoop hit in cache at distance 1, hardware data prefetch.
0x8415	DSNP_HIT_N2_HWPRF	Snoop hit in cache at distance 2, hardware data prefetch.
0x8416	DSNP_HIT_N3_HWPRF	Snoop hit in cache at distance 3, hardware data prefetch.
0x8417	DSNP_HIT_N4_HWPRF	Snoop hit in cache at distance 4, hardware data prefetch.
0x8418	ISNP_HIT_N1_PRFM	Snoop hit in cache at distance 1, software instruction prefetch.
0x8419	ISNP_HIT_N2_PRFM	Snoop hit in cache at distance 2, software instruction prefetch.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x841A	ISNP_HIT_N3_PRFM	Snoop hit in cache at distance 3, software instruction prefetch.
0x841B	ISNP_HIT_N4_PRFM	Snoop hit in cache at distance 4, software instruction prefetch.
0x841C	ISNP_HIT_N1_HWPRF	Snoop hit in cache at distance 1, hardware instruction prefetch.
0x841D	ISNP_HIT_N2_HWPRF	Snoop hit in cache at distance 2, hardware instruction prefetch.
0x841E	ISNP_HIT_N3_HWPRF	Snoop hit in cache at distance 3, hardware instruction prefetch.
0x841F	ISNP_HIT_N4_HWPRF	Snoop hit in cache at distance 4, hardware instruction prefetch.
0x8420	DSNP_HIT_N1_PRF	Snoop hit in cache at distance 1, data prefetch.
0x8421	DSNP_HIT_N2_PRF	Snoop hit in cache at distance 2, data prefetch.
0x8422	DSNP_HIT_N3_PRF	Snoop hit in cache at distance 3, data prefetch.
0x8423	DSNP_HIT_N4_PRF	Snoop hit in cache at distance 4, data prefetch.
0x8424	ISNP_HIT_N1_PRF	Snoop hit in cache at distance 1, instruction prefetch.
0x8425	ISNP_HIT_N2_PRF	Snoop hit in cache at distance 2, instruction prefetch.
0x8426	ISNP_HIT_N3_PRF	Snoop hit in cache at distance 3, instruction prefetch.
0x8427	ISNP_HIT_N4_PRF	Snoop hit in cache at distance 4, instruction prefetch.
0x8428	ISNP_HIT_N1	Snoop hit in cache at distance 1, instruction access.
0x8429	ISNP_HIT_N2	Snoop hit in cache at distance 2, instruction access.
0x842A	ISNP_HIT_N3	Snoop hit in cache at distance 3, instruction access.
0x842B	ISNP_HIT_N4	Snoop hit in cache at distance 4, instruction access.
0x842C	DSNP_HIT_N1	Snoop hit in cache at distance 1, data access.
0x842D	DSNP_HIT_N2	Snoop hit in cache at distance 2, data access.
0x842E	DSNP_HIT_N3	Snoop hit in cache at distance 3, data access.
0x842F	DSNP_HIT_N4	Snoop hit in cache at distance 4, data access.
0x8431	ASE_FP_VREDUCE_SPEC	Floating-point operation speculatively executed, Advanced SIMD pairwise or reduction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x8432	SVE_FP_PREDUCE_SPEC	Floating-point operation speculatively executed, SVE pairwise add step or pairwise reduce step.
0x8434	DSNP_HITM_N1_RD	Snoop hit in cache at distance 1, demand data read, modified.
0x8435	DSNP_HITM_N2_RD	Snoop hit in cache at distance 2, demand data read, modified.
0x8436	DSNP_HITM_N3_RD	Snoop hit in cache at distance 3, demand data read, modified.
0x8437	DSNP_HITM_N4_RD	Snoop hit in cache at distance 4, demand data read, modified.
0x8438	DSNP_HITM_N1_WR	Snoop hit in cache at distance 1, demand data write, modified.
0x8439	DSNP_HITM_N2_WR	Snoop hit in cache at distance 2, demand data write, modified.
0x843A	DSNP_HITM_N3_WR	Snoop hit in cache at distance 3, demand data write, modified.
0x843B	DSNP_HITM_N4_WR	Snoop hit in cache at distance 4, demand data write, modified.
0x843C	DSNP_HITM_N1_RW	Snoop hit in cache at distance 1, demand data access, modified.
0x843D	DSNP_HITM_N2_RW	Snoop hit in cache at distance 2, demand data access, modified.
0x843E	DSNP_HITM_N3_RW	Snoop hit in cache at distance 3, demand data access, modified.
0x843F	DSNP_HITM_N4_RW	Snoop hit in cache at distance 4, demand data access, modified.
0x8440	ASE_FP_SP_MIN_SPEC	Advanced SIMD data processing operation speculatively executed, smallest type is single-precision floating-point.
0x8442	ASE_FP_HP_MIN_SPEC	Advanced SIMD data processing operation speculatively executed, smallest type is half-precision floating-point.
0x8443	ASE_FP_BF16_MIN_SPEC	Advanced SIMD data processing operation speculatively executed, smallest type is BFloat16 floating-point.
0x8444	ASE_FP_FP8_MIN_SPEC	Advanced SIMD data processing operation speculatively executed, smallest type is 8-bit floating-point.
0x8448	ASE_SVE_FP_SP_MIN_SPEC	Advanced SIMD data processing or SVE data processing operation speculatively executed, smallest type is single-precision floating-point.
0x844A	ASE_SVE_FP_HP_MIN_SPEC	Advanced SIMD data processing or SVE data processing operation speculatively executed, smallest type is half-precision floating-point.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x844B	ASE_SVE_FP_BF16_MIN_SPEC	Advanced SIMD data processing or SVE data processing operation speculatively executed, smallest type is BFloat16 floating-point.
0x844C	ASE_SVE_FP_FP8_MIN_SPEC	Advanced SIMD data processing or SVE data processing operation speculatively executed, smallest type is 8-bit floating-point.
0x8450	SE_FP_SP_MIN_SPEC	Advanced SIMD, SVE, or SME data processing operation speculatively executed, smallest type is single-precision floating-point.
0x8452	SE_FP_HP_MIN_SPEC	Advanced SIMD, SVE, or SME data processing operation speculatively executed, smallest type is half-precision floating-point.
0x8453	SE_FP_BF16_MIN_SPEC	Advanced SIMD, SVE, or SME data processing operation speculatively executed, smallest type is BFloat16 floating-point.
0x8454	SE_FP_FP8_MIN_SPEC	Advanced SIMD, SVE, or SME data processing operation speculatively executed, smallest type is 8-bit floating-point.
0x8458	SME_FP_SP_MIN_SPEC	SME data processing operation speculatively executed, smallest type is single-precision floating-point.
0x845A	SME_FP_HP_MIN_SPEC	SME data processing operation speculatively executed, smallest type is half-precision floating-point.
0x845B	SME_FP_BF16_MIN_SPEC	SME data processing operation speculatively executed, smallest type is BFloat16 floating-point.
0x845C	SME_FP_FP8_MIN_SPEC	SME data processing operation speculatively executed, smallest type is 8-bit floating-point.
0x8460	SVE_FP_SP_MIN_SPEC	SVE data processing operation speculatively executed, smallest type is single-precision floating-point.
0x8462	SVE_FP_HP_MIN_SPEC	SVE data processing operation speculatively executed, smallest type is half-precision floating-point.
0x8463	SVE_FP_BF16_MIN_SPEC	SVE data processing operation speculatively executed, smallest type is BFloat16 floating-point.
0x8464	SVE_FP_FP8_MIN_SPEC	SVE data processing operation speculatively executed, smallest type is 8-bit floating-point.
0x846B	FP_BF16_FIXED_OPS_SPEC	Non-scalable element arithmetic operations speculatively executed, largest type is BFloat16 floating-point.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number	Mnemonic	Description
0x8470	FP_SP_MIN_SPEC	Floating-point operation speculatively executed, smallest type is single-precision floating-point.
0x8472	FP_HP_MIN_SPEC	Floating-point operation speculatively executed, smallest type is half-precision floating-point.
0x8473	FP_BF16_MIN_SPEC	Floating-point operation speculatively executed, smallest type is BFloat16 floating-point.
0x8474	FP_FP8_MIN_SPEC	Floating-point operation speculatively executed, smallest type is 8-bit floating-point.
0x8480	FP_SP_FIXED_MIN_OPS_SPEC	Non-scalable element arithmetic operations speculatively executed, smallest type is single-precision floating-point.
0x8482	FP_HP_FIXED_MIN_OPS_SPEC	Non-scalable element arithmetic operations speculatively executed, smallest type is half-precision floating-point.
0x8483	FP_BF16_FIXED_MIN_OPS_SPEC	Non-scalable element arithmetic operations speculatively executed, smallest type is BFloat16 floating-point.
0x8484	FP_FP8_FIXED_MIN_OPS_SPEC	Non-scalable element arithmetic operations speculatively executed, smallest type is 8-bit floating-point.
	TRB_TRIG	Trace buffer Trigger Event.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D9.3 Common event numbers

The event numbers of the common architectural and microarchitectural events are reserved for the specified events. Each of these event numbers must either:

- Be used for its assigned event.
- Not be used.

However, see Required events.

When an implementation supports monitoring of an event that is assigned a common architectural or microarchitectural event number, Arm strongly recommends that it uses that number for the event. However, software might encounter implementations where an event assigned a number in this range is monitored using an event number from an IMPLEMENTATION DEFINED range.

Note

Arm might define other common architectural and microarchitectural event numbers. This is one reason why software must not assume that an event with an assigned common architectural or microarchitectural event number is never monitored using an event number from the IMPLEMENTATION DEFINED range.

It is IMPLEMENTATION DEFINED which events, including Common events, are generated by IMPLEMENTATION DEFINED extensions to the architecture, including accesses to IMPLEMENTATION DEFINED System registers and IMPLEMENTATION DEFINED System instructions. However, the functionality of the IMPLEMENTATION DEFINED extension must be appropriate for the generated events.

The common events are described in the following sections:

- Common architectural events.
- Common microarchitectural events.

The supported common architectural and microarchitectural events in the ranges `0x0000-0x003F` and `0x4000-0x403F` are discoverable to software through:

- PMCEID0_EL0.
- PMCEID1_EL0.

Arm recommends that the value of 0 is used for the PMCEID0_EL0 or PMCEID1_EL0 bit corresponding to any event that an implementation never generates, even if the implementation is considered to support but never count the event.

Note

For example, if an implementation never generates the L1D_CACHE_ALLOCATE event, event 31, Arm recommends that PMCEID0_EL0[31] is RAZ.

However, for some implementations, an event in the common events range might be generated by the system, meaning behavior can vary between systems. In such a case, the corresponding PMCEID*n*_EL0 bit might be RAO.

Event numbers that Table D9-2 shows as allocated for common architectural and microarchitectural events that are not described in Common architectural events and Common microarchitectural events are reserved. Future revisions of this Manual, or of the architecture, might assign these reserved values to additional common events. Events that do not require additional features in the PMU can be implemented retrospectively, meaning an implementation of a particular version of the PMU specification might support common events that are first defined in a later version of the PMU specification.

Note

The requirement that an event that is implemented retrospectively *does not require additional features in the PMU* means that it must be possible to represent the event *n* the PMEVTYPER<n>_EL0.evtCount field. This means, for example, that an implementation with a 10-bit PMEVTYPER<n>_EL0.evtCount field can only implement events with event numbers `0x0000-0x03FF`.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D9.3.1 Common architectural events

The Common architectural events are:

`0x0000`, **SW_INCR, Instruction architecturally executed, Condition code check pass, software increment**

The counter counts each write to the PMSWINC_EL0 register for each implemented event counter <n> if PMEVTYPER<n>_EL0.evtCount is `0x0000` then the counter counts each `MSR` write to PMSWINC_EL0 with bit [n] set to 1.

If the PE performs two architecturally executed writes to the PMSWINC_EL0 register without an intervening Context synchronization event, then the counter is incremented twice.

Note

In a multithreaded implementation, if the Effective value of PMEVTYPER<n>_EL0.MT is 1, then the counter counts writes by all PEs that have the same affinity at level 1 and above.

PMCEID0_EL0[0] reads as 1 if this event is implemented and 0 otherwise.

`0x0006`, **LD_RETIRED, Instruction architecturally executed, Condition code check pass, load**

The counter counts each architecturally-executed memory-reading instruction counted by INST_RETIRED.

It is IMPLEMENTATION DEFINED whether the prefetch instructions counted by PRF_RETIRED are counted as Memory-reading instructions:

- If the prefetch instructions are counted as Memory-reading instructions, then they are counted by LD_RETIRED and LD_ANY_RETIRED.
- Otherwise, if the prefetch instructions are not counted as Memory-reading instructions by LD_RETIRED, then it is further IMPLEMENTATION DEFINED which one of the following applies:
  - They are counted as data-processing instructions.
  - They are counted by LDST_ANY_RETIRED.

Arm recommends that if a prefetch instruction is not implemented as a NOP and the PRF_RETIRED event is not implemented, then the instruction is counted as a Memory-reading instruction.

PMCEID0_EL0[6] reads as 1 if this event is implemented and 0 otherwise.

`0x0007`, **ST_RETIRED, Instruction architecturally executed, Condition code check pass, store**

The counter counts each architecturally-executed memory-writing instruction counted by INST_RETIRED.

The counter does not count a Store-Exclusive instruction that fails.

PMCEID0_EL0[7] reads as 1 if this event is implemented and 0 otherwise.

`0x0008`, **INST_RETIRED, Instruction architecturally executed**

The counter counts each architecturally-executed instruction.

PMCEID0_EL0[8] reads as 1 if this event is implemented and 0 otherwise.

`0x0009`, **EXC_TAKEN, Exception taken**

The counter counts each exception taken.

The counter counts each exception taken.

Note

The counter counts the PE exceptions described in Exception entry.

PMCEID0_EL0[9] reads as 1 if this event is implemented and 0 otherwise.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x000A`, **EXC_RETURN, Instruction architecturally executed, Condition code check pass, exception return**

The counter counts each architecturally-executed exception return instruction.

Instructions defined in the following sections are counted as exception return instructions:

- For an exception return from an Exception level using AArch64, Exception return.
- For an exception return from an Exception level using AArch32, Exception return instructions.

It is CONSTRAINED UNPREDICTABLE whether the execution of an exception return instruction is counted if the instruction sets PSTATE.IL and does not generate an exception return.

PMCEID0_EL0[10] reads as 1 if this event is implemented and 0 otherwise.

`0x000B`, **CID_WRITE_RETIRED, Instruction architecturally executed, Condition code check pass, write to CONTEXTIDR**

The counter counts each `MSR` write to CONTEXTIDR_EL1.

If the PE performs two architecturally-executed writes to CONTEXTIDR without an intervening Context synchronization event, it is CONSTRAINED UNPREDICTABLE whether the first write is counted.

The counter counts only writes to these named registers. For example:

- When FEAT_Debugv8p2 is implemented, the counter does not count writes using the register name CONTEXTIDR_EL2.

PMCEID0_EL0[11] reads as 1 if this event is implemented and 0 otherwise.

`0x000C`, **PC_WRITE_RETIRED, Instruction architecturally executed, Condition code check pass, Software change of the PC**

The counter counts each architecturally-executed Software change of the PC that is taken.

This includes all:

- Branch instructions.
- Memory-reading instructions that explicitly write to the PC.
- Data-processing instructions that explicitly write to the PC.
- Exception return instructions.

Note

Conditional branches are only counted if the branch is taken.

It is IMPLEMENTATION DEFINED whether the following instructions are included as a Software change of the PC:

- A `BRK` instruction.
- An UNDEFINED instruction that generates an exception.
- An exception-generating instruction, `SVC`, `HVC`, and `SMC`, that generates an exception.
- A Context synchronization barrier instruction, `ISB`.

The counter does not increment for exceptions other than those explicitly identified in these lists.

PMCEID0_EL0[12] reads as 1 if this event is implemented and 0 otherwise.

`0x000D`, **BR_IMMED_RETIRED, Branch instruction architecturally executed, immediate**

The counter counts each architecturally-executed immediate branch instruction.

The following instructions are counted as immediate branch instructions:

- For AArch64 state, the following instructions:
  - `B <label>`.
  - `B.cond <label>`.
  - `BL <label>`.
  - `CBZ <Rn>, <label>`.
  - `CBNZ <Rn>, <label>`.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- `TBZ <Rn>, <label>`.
- `TBNZ <Rn>, <label>`.

Note

Conditional branches are always counted, regardless of whether the branch is taken or not taken.

If the Context synchronization barrier instruction ISB is counted as a Software change of the PC instruction by PC_WRITE_RETIRED, then it is IMPLEMENTATION DEFINED whether ISB is counted as an immediate branch instruction.

PMCEID0_EL0[13] reads as 1 if this event is implemented and 0 otherwise.

`0x000E`, **BR_RETURN_RETIRED, Branch instruction architecturally executed, procedure return, taken**

The counter counts each architecturally executed procedure return instruction.

The following instructions are counted as procedure return instructions:

- For AArch64 state, the following instructions:
  - `RET`.
  - If FEAT_PAuth is implemented, `RETAA` and `RETAB`.

PMCEID0_EL0[14] reads as 1 if this event is implemented and 0 otherwise.

`0x000F`, **UNALIGNED_LDST_RETIRED, Instruction architecturally executed, Condition code check pass, unaligned load or store**

The counter counts each memory-reading instruction or memory-writing instruction access that would generate an Alignment fault when Alignment fault checking is enabled.

The counter does not count accesses that would generate an SP alignment fault exception if the applicable stack pointer alignment check is enabled, unless that access would also generate an Alignment fault Data Abort exception if Alignment fault checking is enabled.

It is IMPLEMENTATION DEFINED and might be UNPREDICTABLE whether this event counts accesses that generate an exception, including accesses that do generate Alignment fault Data Abort exceptions.

PMCEID0_EL0[15] reads as 1 if this event is implemented and 0 otherwise.

`0x001C`, **TTBR_WRITE_RETIRED, Instruction architecturally executed, Condition code check pass, write to TTBR**

The counter counts each `MSR` write to TTBR0_EL1 and TTBR1_EL1 in AArch64 state.

The counter counts `MSR` writes to TTBR0_EL1 and TTBR1_EL1.

If the PE executes two writes to the same TTBR, without an intervening Context synchronization event, it is CONSTRAINED UNPREDICTABLE whether the first write to the TTBR, is counted.

The counter counts only writes to these registers by name. For example:

- The counter does not count writes using the register names TTBR0_EL2 and VTTBR_EL2.

PMCEID0_EL0[28] reads as 1 if this event is implemented and 0 otherwise.

`0x001E`, **CHAIN, Chain a pair of event counters**

Even-numbered counters never increment as a result of this event.

For an odd-numbered counter <n+1>, the counter increments when an event increments the preceding even-numbered counter <n> on the same PE causing unsigned overflow of bits [31:0] of the event counter <n>.

This means the CHAIN event can be used to link the odd-numbered counter with the preceding even-numbered counter to provide a 64-bit counter.

If EL2 is implemented and <n+1> is equal to the Effective value of HDCR.HPMN, then it is UNPREDICTABLE whether the counter counts.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

- The CHAIN event allows software to use the N event counters as N 32-bit counters, N÷2 64-bit counters, or a mixture of 32-bit counters and 64-bit counters.
- The CHAIN event only counts overflows from the preceding even-numbered counter on the same PE. This means it ignores the Effective value of PMEVTYPER<n>_EL0.MT.
- The architecture does not provide atomic access to a pair of counters.

To filter the Exception levels and Security states in which the event is counted, software:

- Programs PMEVTYPER<n>_EL0 to count the event in the required conditions.
- Programs PMEVTYPER<n+1>_EL0 to count the CHAIN event in all Exception levels and states.

The PE might ignore the filter settings for the CHAIN event and behave as if they are set to count in all Exception levels and states. If software does not program the event in this way, the count becomes UNPREDICTABLE.

The architecture does not define the latency between the low counter overflowing and the high counter incrementing the CHAIN event. There is no requirement for updates to occur synchronously, but software reading or enabling the counter pair by reading the low counter first and the high counter second, with an intervening Context synchronization event, will not observe the low counter incrementing and overflowing for the event and the high counter not incrementing for the resulting CHAIN event. This means that the ISB executed after reading the low counter ensures the completion of the update of the high counter by the CHAIN event.

PMCEID0_EL0[30] reads as 1 if this event is implemented and 0 otherwise.

`0x0021`**, BR_RETIRED, Instruction architecturally executed, branch**

The counter counts each branch instruction on the architecturally executed path that would incur cost if mispredicted.

It is IMPLEMENTATION DEFINED whether the counter increments for any or all of:

- Unconditional direct branch instructions.
- Exception-generating instructions.
- Exception return instructions.
- Context synchronization instructions.

The counter counts all other branch instructions, memory-reading instructions and data-processing instructions that explicitly write to the PC, at retirement.

Note

Conditional branches are always counted, regardless of whether the branch is taken.

Arm recommends that BR_RETIRED counts Unconditional direct branch instructions and Exception return instructions.

PMCEID1_EL0[1] reads as 1 if this event is implemented and 0 otherwise.

`0x0081`**, EXC_UNDEF, Exception taken, other synchronous**

The counter counts each exception counted by EXC_TAKEN that is Taken locally and is not counted as any of the following:

- Exception taken, Supervisor Call (EXC_SVC).
- Exception taken, Secure Monitor Call (EXC_SMC).
- Exception taken, Hypervisor Call (EXC_HVC).
- Exception taken, Instruction Abort (EXC_PABORT).
- Exception taken, Data Abort or SError (EXC_DABORT).
- Exception taken, IRQ (EXC_IRQ).
- Exception taken, FIQ (EXC_FIQ).

`0x0082`**, EXC_SVC, Exception taken, Supervisor Call**

The counter counts each Supervisor Call exception counted by EXC_TAKEN that is Taken locally.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x0083`, **EXC_PABORT, Exception taken, Instruction Abort**

The counter counts each Instruction Abort exception counted by EXC_TAKEN that is Taken locally.

`0x0084`, **EXC_DABORT, Exception taken, Data Abort or SError**

The counter counts each Data Abort, SError interrupt, or virtual SError interrupt exception counted by EXC_TAKEN that is Taken locally.

`0x0086`, **EXC_IRQ, Exception taken, IRQ**

The counter counts each IRQ or virtual IRQ exception counted by EXC_TAKEN that is Taken locally.

`0x0087`, **EXC_FIQ, Exception taken, FIQ**

The counter counts each FIQ or virtual FIQ exception counted by EXC_TAKEN that is Taken locally.

`0x0088`, **EXC_SMC, Exception taken, Secure Monitor Call**

The counter counts each Secure Monitor Call exception counted by EXC_TAKEN.

The counter does not count SMC instructions that generate other exceptions, including Trap exceptions and Undefined Instruction exceptions.

`0x008A`, **EXC_HVC, Exception taken, Hypervisor Call**

The counter counts each Hypervisor Call exception counted by EXC_TAKEN.

The counter does not count HVC instructions that are UNDEFINED.

`0x008B`, **EXC_TRAP_PABORT, Exception taken, Instruction Abort not Taken locally**

The counter counts each Instruction Abort exception counted by EXC_TAKEN that is not Taken locally.

`0x008C`, **EXC_TRAP_DABORT, Exception taken, Data Abort or SError not Taken locally**

The counter counts each Data Abort, SError interrupt, or virtual SError interrupt exception counted by EXC_TAKEN, that is not Taken locally.

`0x008D`, **EXC_TRAP_OTHER, Exception taken, other traps not Taken locally**

The counter counts each exception counted by EXC_TAKEN that is not Taken locally and not counted as any of the following:

- Exception taken, Secure Monitor Call (EXC_SMC).
- Exception taken, Hypervisor Call (EXC_HVC).
- Exception taken, Instruction Abort not Taken locally (EXC_TRAP_PABORT).
- Exception taken, Data Abort or SError not Taken locally (EXC_TRAP_DABORT).
- Exception taken, IRQ not Taken locally (EXC_TRAP_IRQ).
- Exception taken, FIQ not Taken locally (EXC_TRAP_FIQ).

`0x008E`, **EXC_TRAP_IRQ, Exception taken, IRQ not Taken locally**

The counter counts each IRQ or virtual IRQ exception counted by EXC_TAKEN that is not Taken locally.

`0x008F`, **EXC_TRAP_FIQ, Exception taken, FIQ not Taken locally**

The counter counts each FIQ or virtual FIQ exception counted by EXC_TAKEN that is not Taken locally.

`0x400C`, **TRB_WRAP, Trace buffer current write pointer wrapped**

The counter counts each write made by the Trace Buffer Unit to the trace buffer that causes the trace buffer current write pointer to wrap to the trace buffer base pointer.

PMCEID0_EL0[44] reads as 1 if this event is implemented and 0 otherwise.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x4030`, **TSTART_RETIRED, Instruction architecturally executed, outer transaction TSTART**

The counter counts each TSTART instruction counted by INST_RETIRED that starts an outer transaction.

If PMEVTYPER<n>_EL0.T is 1, it is IMPLEMENTATION DEFINED whether or not the TSTART instruction is counted by the counter.

PMCEID1_EL0[48] reads as 1 if this event is implemented and 0 otherwise.

`0x4031`, **TCOMMIT_RETIRED, Instruction architecturally executed, outer transaction TCOMMIT**

The counter counts each TCOMMIT instruction counted by INST_RETIRED that commits an outer transaction.

If PMEVTYPER<n>_EL0.T is 1, it is IMPLEMENTATION DEFINED whether or not the TCOMMIT instruction is counted by the counter.

PMCEID1_EL0[49] reads as 1 if this event is implemented and 0 otherwise.

`0x8000`, **SIMD_INST_RETIRED, Instruction architecturally executed, SIMD**

The counter counts each architecturally executed SIMD instruction.

It is IMPLEMENTATION DEFINED whether scalar loads and stores to SIMD&FP registers other that those listed above are counted.

`0x8001`, **ASE_INST_RETIRED, Instruction architecturally executed, Advanced SIMD**

The counter counts each architecturally executed instruction counted by ASE_SVE_INST_RETIRED that is classified as an Advanced SIMD instruction.

The following instructions are counted as Advanced SIMD instructions:

- For AArch64 state:
  - The SIMD operations listed in Data processing - SIMD and floating-point.
  - If FEAT_Crypto is implemented, the Advanced SIMD Cryptographic instructions. See The Cryptographic Extension in AArch64 state.
  - Advanced SIMD loads and stores.

This includes all instructions that operate on the SIMD&FP registers, except those that are counted as one of the following:

- Integer data-processing instructions.
- Scalar floating-point data-processing instructions.

It is IMPLEMENTATION DEFINED whether Advanced SIMD scalar instructions are counted as Advanced SIMD instructions. This includes Advanced SIMD scalar instructions which operate on floating-point values.

`0x8003`, **ASE_SVE_INST_RETIRED, Instruction architecturally executed, Advanced SIMD or SVE**

The counter counts each architecturally executed instruction counted by SE_INST_RETIRED that is classified as an Advanced SIMD instruction or an SVE instruction.

See ASE_INST_RETIRED and SVE_INST_RETIRED for these classifications.

`0x8107`, **BR_SKIP_RETIRED, Branch instruction architecturally executed, not taken**

The counter counts each conditional Software change of the PC instruction that is not taken.

This includes all:

- Conditional branch instructions.
- Conditional memory-reading instructions that explicitly write to the PC.
- Conditional data-processing instructions that explicitly write to the PC.
- Conditional exception return instructions.

These are the same instructions which, if unconditional, or conditional and taken, are counted by the PC_WRITE_RETIRED event.

It is IMPLEMENTATION DEFINED whether the following instructions are included as a Software change of the PC, which the counter increments for:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- A conditional UNDEFINED instruction that does not generate an exception.
- A conditional exception-generating instruction, SVC, HVC, or SMC, that does not generate an exception.
- A conditional Context synchronization barrier instruction, ISB, that fails its condition code check.

Note

Many of these instructions can only be conditional in the AArch32 instruction sets.

The counter does not increment for exceptions other than those explicitly identified in these lists.

`0x8108`, **BR_IMMED_TAKEN_RETIRED, Branch instruction architecturally executed, immediate, taken**

The counter counts each instruction counted by both BR_IMMED_RETIRED and PC_WRITE_RETIRED.

These are all direct branch instructions on the architecturally executed path, where the branch is taken.

`0x810C`, **BR_INDNR_TAKEN_RETIRED, Branch instruction architecturally executed, indirect excluding procedure return, taken**

The counter counts each instruction counted by both BR_IND_RETIRED and PC_WRITE_RETIRED, that is not counted by BR_RETURN_RETIRED.

These are all indirect branch instructions, excluding return instructions, on the architecturally executed path, where the branch is taken.

`0x811D`, **BR_IND_RETIRED, Instruction architecturally executed, indirect branch**

The counter counts each architecturally executed indirect branch instruction.

Indirect branch instructions are Software change of the PC instructions other than exception-generating instructions and immediate branch instructions. Software change of the PC has the same definition as for the PC_WRITE_RETIRED and BR_SKIP_RETIRED events. Immediate branch instructions are defined by the BR_IMMED_RETIRED event.

Note

Conditional branches are always counted, regardless of whether the branch is taken.

`0x811F`, **BRB_FILTRATE, Branch Record captured**

The counter counts each valid Branch record captured in the branch record buffer.

Branch records that are not captured because they are removed by filtering are not counted.

When BRB_FILTRATE is generated for an exception or an exception return, it is an Exception-related event.

It is CONSTRAINED UNPREDICTABLE whether the counter counts Branch records injected by a BRB INJ instruction.

If counting this event causes unsigned overflow of the event counter counting the event and this in turn causes a BRBE freeze event then:

- The Branch record for the operation that generated the event will be generated and captured in the Branch record buffer.
- It is CONSTRAINED UNPREDICTABLE whether the Branch Record Buffer Extension generates Branch records for other operations in program order after the operation that generated the event that would otherwise be generated when generation of Branch records is not Paused.

Arm recommends that implementations minimize capture of additional branches.

`0x8179`, **BRNL_INDNR_TAKEN_RETIRED, Branch instruction architecturally executed, indirect branch without link excluding procedure return, taken**

The counter counts each branch counted by BRNL_IND_TAKEN_RETIRED that is not counted by BR_RETURN_RETIRED.

These are all indirect branch without link excluding procedure return instructions on the architecturally executed path, where the branch is taken.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x817A`, **BL_TAKEN_RETIRED, Branch instruction architecturally executed, branch with link, taken**

The counter counts each Software change of the PC counted by PC_WRITE_RETIRED that calls a subroutine and sets LR to return address.

These are all branch with link instructions on the architecturally executed path, where the branch is taken.

`0x817B`, **BRNL_TAKEN_RETIRED, Branch instruction architecturally executed, branch without link, taken**

The counter counts each Software change of the PC counted by PC_WRITE_RETIRED that is not counted by BL_TAKEN_RETIRED.

These are all branch without link instructions on the architecturally executed path, where the branch is taken.

`0x817C`, **BL_IND_TAKEN_RETIRED, Branch instruction architecturally executed, indirect branch with link, taken**

The counter counts each branch counted by both BL_TAKEN_RETIRED and BR_IND_TAKEN_RETIRED.

These are all indirect branch with link instructions on the architecturally executed path, where the branch is taken.

`0x817D`, **BRNL_IND_TAKEN_RETIRED, Branch instruction architecturally executed, indirect branch without link, taken**

The counter counts each branch counted by both BRNL_TAKEN_RETIRED and BR_IND_TAKEN_RETIRED.

These are all indirect branch without link instructions on the architecturally executed path, where the branch is taken.

`0x817E`, **BL_IMMED_TAKEN_RETIRED, Branch instruction architecturally executed, direct branch with link, taken**

The counter counts each branch counted by both BL_TAKEN_RETIRED and BR_IMMED_TAKEN_RETIRED.

These are all direct branch with link instructions on the architecturally executed path, where the branch is taken.

`0x817F`, **BRNL_IMMED_TAKEN_RETIRED, Branch instruction architecturally executed, direct branch without link, taken**

The counter counts each branch counted by both BRNL_TAKEN_RETIRED and BR_IMMED_TAKEN_RETIRED.

These are all direct branch without link instructions on the architecturally executed path, where the branch is taken.

`0x8180`, **BR_UNCOND_RETIRED, Branch instruction architecturally executed, unconditional branch**

The counter counts each Software change of the PC counted by BR_RETIRED that is not counted by BR_COND_RETIRED.

These are all unconditional branch instructions on the architecturally executed path.

`0x8181`, **BR_COND_RETIRED, Branch instruction architecturally executed, conditional branch**

The counter counts each Software change of the PC counted by BR_RETIRED that is a conditional branch.

These are all conditional branch instructions on the architecturally executed path.

In the A64 instruction set, conditional branches are:

- `B.cond`, `CBNZ`, `CBZ`, `TBNZ`, and `TBZ`.

This includes conditional branch instructions with the `AL` or `NV` condition code.

`0x8182`, **BR_COND_TAKEN_RETIRED, Branch instruction architecturally executed, conditional branch, taken**

The counter counts each Software change of the PC counted by both BR_COND_RETIRED and PC_WRITE_RETIRED.

These are all conditional branch instructions on the architecturally executed path, where the branch is taken.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x82A4`, **ASE_SVE_RETIRED, Instruction architecturally executed, Advanced SIMD data processing or SVE data processing**

The counter counts each architecturally executed instruction counted by INST_RETIRED that is classified as an Advanced SIMD data-processing instruction or an SVE data-processing instruction.

See ASE_RETIRED and SVE_RETIRED for these classifications.

`0x82A8`, **LD_ANY_RETIRED, Instruction architecturally executed, load**

The counter counts each architecturally executed instruction counted by LDST_ANY_RETIRED that is classified as a Memory-reading instruction.

It is IMPLEMENTATION DEFINED whether the prefetch instructions counted by PRF_RETIRED are counted as Memory-reading instructions:

- If the prefetch instructions are counted as Memory-reading instructions, then they are counted by LD_RETIRED and LD_ANY_RETIRED.
- Otherwise, if the prefetch instructions are not counted as Memory-reading instructions by LD_ANY_RETIRED, then it is further IMPLEMENTATION DEFINED which one of the following applies:
  - They are counted as data-processing instructions.
  - They are counted by LDST_ANY_RETIRED.

Arm recommends that if a prefetch instruction is not implemented as a NOP and the PRF_RETIRED event is not implemented, then the instruction is counted as a Memory-reading instruction.

`0x82A9`, **ST_ANY_RETIRED, Instruction architecturally executed, store**

The counter counts each architecturally executed instruction counted by LDST_ANY_RETIRED that is classified as a Memory-writing instruction.

`0x82AA`, **LDST_ANY_RETIRED, Instruction architecturally executed, load or store**

The counter counts each architecturally executed instruction counted by INST_RETIRED that is classified as a Memory-reading instruction or a Memory-writing instruction.

See LD_ANY_RETIRED and ST_ANY_RETIRED for these classifications.

If the PRF_RETIRED event is implemented and the prefetch instructions counted by PRF_RETIRED are not counted as Memory-reading instructions by LD_RETIRED and LD_ANY_RETIRED, then it is IMPLEMENTATION DEFINED whether these instructions are counted by LDST_ANY_RETIRED. Arm recommends that these instructions are counted by LDST_ANY_RETIRED.

`0x82AB`, **DP_RETIRED, Instruction architecturally executed, integer data processing**

The counter counts each architecturally executed instruction counted by INST_RETIRED that is classified as an integer data-processing instruction.

An integer data-processing instruction is any instruction that is not counted as any of the following:

- A load or store instruction, counted by LD_RETIRED or ST_RETIRED.
- A Software change of PC instruction, counted by PC_WRITE_RETIRED.
- A scalar floating-point data processing instruction, counted by VFP_RETIRED.
- An Advanced SIMD or SVE data processing instruction, counted by ASE_SVE_RETIRED.
- A cryptographic instruction, counted by CRYPTO_RETIRED.

That is, the following instructions are counted as integer data-processing instructions:

- In AArch64 state:
  - Instructions from Data processing - immediate.
  - Instructions from Data processing - register.
  - Instructions from System register instructions.
  - Instructions from ‘Instructions with register argument’.
  - Instructions from System instructions other than Memory-writing instructions.
  - Instructions from Hint instructions.
  - Instructions from Floating-point move (register) that transfer data between a general-purpose register and a SIMD&FP register without conversion: `FMOV` (general).

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Instructions from SIMD move that transfer data between a general-purpose register and an element or elements in a SIMD&FP register: `DUP` (general), `SMOV`, `UMOV`, and `INS` (general). This includes the aliases `MOV` (from general) and `MOV` (to general).
- `VDUP` (general-purpose register).
- All `VMOV` instructions that transfer data between a general-purpose register and a SIMD&FP register.
- `VMRS` and `VMSR`.

`0x82AC`, **ASE_RETIRED, Instruction architecturally executed, Advanced SIMD data processing**

The counter counts each architecturally executed instruction counted by ASE_SVE_RETIRED that is classified as an Advanced SIMD data-processing instruction.

The following instructions are counted as Advanced SIMD data-processing instructions:

- For AArch64 state:
  - The SIMD operations listed in Data processing - SIMD and floating-point.
  - Advanced SIMD scalar instructions, including those which operate on floating-point values.
  - If FEAT_Crypto is implemented, the Advanced SIMD `PMULL` and `PMULL2` (1Q variants) instructions. See The Cryptographic Extension in AArch64 state.

This includes all instructions that operate on the SIMD&FP registers, except those that are counted as one of the following:

- Integer data-processing instructions.
- Scalar floating-point data-processing instructions.
- Load or store instructions.
- Cryptographic data-processing instructions counted by CRYPTO_RETIRED. CRYPTO_RETIRED does not count the Cryptographic data-processing instructions included above.

`0x82AD`, **VFP_RETIRED, Instruction architecturally executed, scalar floating-point data processing**

The counter counts each architecturally executed instruction counted by INST_RETIRED that is classified as a scalar floating-point instruction.

The following instructions are counted as scalar floating-point data-processing instructions:

- In AArch64 state:
  - The scalar floating-point instructions from Data processing - SIMD and floating-point. Only the scalar instructions are counted. SIMD floating-point instructions are not counted.
  - Instructions that take both an integer register and a floating-point register argument and perform a type conversion (to/from integer or to/from fixed-point): `FCVT{<mode>}`, `UCVTF`, and `SCVTF`.

This includes all scalar instructions that operate on the SIMD&FP registers as floating-point values, except those that are counted as one of the following:

- SIMD scalar instructions.
- Integer data-processing instructions.
- Load or store instructions.

`0x82AF`, **CRYPTO_RETIRED, Instruction architecturally executed, cryptographic data processing**

The counter counts each architecturally executed instruction counted by INST_RETIRED that is classified as a cryptographic instruction.

The following instructions are counted as cryptographic data-processing instructions:

- In AArch64 state, instructions from The Cryptographic Extension in AArch64 state other than the include Advanced SIMD `PMULL` and `PMULL2` (1Q variants) instructions.

`0x82B3`, **PRF_RETIRED, Instruction architecturally executed, prefetch**

The counter counts each architecturally executed prefetch instruction counted by INST_RETIRED.

`0x8359`, **SE_RETIRED, Instruction architecturally executed, Advanced SIMD, SVE, or SME data processing**

The counter counts each architecturally executed instruction counted by INST_RETIRED that is classified as an Advanced SIMD, SVE, or SME data-processing instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

See ASE_SVE_RETIRED and SME_RETIRED for these classifications.

`0x835B`**, SE_INST_RETIRED, Instruction architecturally executed, Advanced SIMD, SVE, or SME**

The counter counts each architecturally executed instruction counted by INST_RETIRED that is classified as an Advanced SIMD, SVE, or SME instruction.

See ASE_SVE_INST_RETIRED and SME_INST_RETIRED for these classifications.

TRB_TRIG, Trace buffer Trigger Event

The counter counts each trace buffer Trigger Event.

It is IMPLEMENTATION DEFINED whether this event can be counted by the PMU.

PMCEID0_EL0[46] reads as 1 if this event is implemented and can be counted by the PMU, and 0 otherwise.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D9.3.2 Common microarchitectural events

The Common microarchitectural events are:

`0x0001`, **L1I_CACHE_REFILL, Level 1 instruction cache refill**

The counter counts each access counted by L1I_CACHE that causes a refill of the Level 1 instruction or unified cache from outside of the Level 1 instruction or unified cache.

A refill includes any access that causes data to be fetched from outside of the Level 1 caches, even if the data is ultimately not allocated into the Level 1 instruction cache. For example, data might be fetched into a buffer but then discarded, rather than being allocated into a cache. These buffers are treated as part of the cache.

For example, the counter counts accesses to the Level 1 instruction cache that cause a refill that is satisfied by fetching data from memory, a Level 2 cache, or a Level 1 cache of another PE.

The counter does not count accesses that miss in the cache but are satisfied by the refill of a previous miss and do not cause a new refill, even if that previous refill is not complete at the time of the miss.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID0_EL0[1] reads as 1 if this event is implemented and 0 otherwise.

`0x0002`, **L1I_TLB_REFILL, Level 1 instruction TLB refill**

The counter counts each Instruction memory access counted by L1I_TLB that causes a TLB refill of the Level 1 instruction TLB.

This includes each Instruction memory access that causes a memory access due to a translation table walk or an access to another level of TLB caching.

The counter does not count the access if any of the following are true:

- The access is due to a TLB maintenance instruction.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the access if any of the following are true:

- The access misses in the TLB and generates a translation table walk, but the result is not allocated into the TLB for any reason other than a Translation fault.
- The access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID0_EL0[2] reads as 1 if this event is implemented and 0 otherwise.

`0x0003`, **L1D_CACHE_REFILL, Level 1 data cache refill**

The counter counts each access counted by L1D_CACHE that causes a refill of the Level 1 data or unified cache from outside of the Level 1 data or unified cache.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A refill includes any access that causes data to be fetched from outside of the Level 1 caches, even if the data is ultimately not allocated into the Level 1 data cache. For example, data might be fetched into a buffer but then discarded, rather than being allocated into a cache. These buffers are treated as part of the cache.

For example, the counter counts accesses to the Level 1 data cache that cause a refill that is satisfier by fetching data from memory, a Level 2 cache, or a Level 1 cache of another PE.

The counter does not count accesses that:

- Miss in the cache but are satisfied by the refill of a previous miss and do not cause a new refill, even if that previous refill is not complete at the time of the miss.
- Miss in the cache but do not generate a refill, such as a write through the cache.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID0_EL0[3] reads as 1 if this event is implemented and 0 otherwise.

`0x0004`**, L1D_CACHE, Level 1 data cache access**

The counter counts each Memory-read operation or Memory-write operation that causes a cache access to at least the Level 1 data or unified cache.

Each access to a cache line is counted including the multiple accesses caused by single instructions such as `LDM` or `STM`. Each access to other Level 1 data or unified memory structures, for example refill buffers, write buffers, and write-back buffers, is also counted.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

If FEAT_PMUv3p4 is implemented, accesses that only update the cache status information for a cache entry without accessing the content of the cache entry are not counted.

If FEAT_PMUv3p4 is not implemented, it is IMPLEMENTATION DEFINED whether accesses that only update the cache status information for a cache entry without accessing the content of the cache entry are counted.

An example of cache status information is whether the cached data is held in an exclusive or shared state.

When the L1D_CACHE_RW event is implemented:

- If the L1D_CACHE_PRFM event is implemented, accesses to the Level 1 data cache due to a prefetch instruction are counted. Otherwise, these accesses are not counted.
- If the L1D_CACHE_HWPRF event is implemented, accesses to the Level 1 data cache due to a hardware prefetcher are counted. Otherwise these accesses are not counted.

When the L1D_CACHE_RW event is not implemented, it IMPLEMENTATION DEFINED whether accesses to the Level 1 data cache due to prefetch instructions or due to a hardware prefetcher are counted.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID0_EL0[4] reads as 1 if this event is implemented and 0 otherwise.

`0x0005`**, L1D_TLB_REFILL, Level 1 data TLB refill**

The counter counts each access counted by L1D_TLB that causes a TLB refill of the Level 1 data or unified TLB.

This includes each access that causes a memory access due to a translation table walk or an access to another level of TLB caching.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The counter does not count the access if any of the following are true:

- The access is due to a TLB maintenance instruction.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the access if any of the following are true:

- The access misses in the TLB and generates a translation table walk, but the result is not allocated into the TLB for any reason other than a Translation fault.
- The access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID0_EL0[5] reads as 1 if this event is implemented and 0 otherwise.

`0x0010`**, BR_MIS_PRED, Branch instruction speculatively executed, mispredicted or not predicted**

The counter counts each correction to the predicted program flow that occurs because of a misprediction from, or no prediction from, the branch prediction resources and that relates to instructions that the branch prediction resources are capable of predicting.

If no program-flow prediction resources are implemented, Arm recommends that the counter counts all branches that are not taken.

PMCEID0_EL0[16] reads as 1 if this event is implemented and 0 otherwise.

`0x0011`**, CPU_CYCLES, Cycle**

The counter increments on every cycle.

All counters are subject to changes in clock frequency, including when a `WFI` or `WFE` instruction stops the clock. This means that it is CONSTRAINED UNPREDICTABLE whether or not CPU_CYCLES continues to increment when the clocks are stopped by `WFI` and `WFE` instructions.

Note

Unlike PMCCNTR_EL0, this count is not affected by the cycle counter controls:

- The counter is not incremented in prohibited regions, so is not affected by PMCR_EL0.DP.
- The counter increments on every cycle, regardless of the value of PMCR_EL0.D.
- The counter is reset when event counters are reset by PMCR_EL0.P, never by PMCR_EL0.C.

In a multithreaded implementation:

- If the Effective value of PMEVTYPER<n>_EL0.MT for the event counter is 0, then the counter counts each Processor cycle when the counting PE is active.
- If the Effective value of PMEVTYPER<n>_EL0.MT for the event counter is 1, then the counter counts each Processor cycle.

PMCEID0_EL0[17] reads as 1 if this event is implemented and 0 otherwise.

`0x0012`**, BR_PRED, Predictable branch instruction speculatively executed**

The counter counts each branch or other change in the program flow that the branch prediction resources are capable of predicting.

If all branches are subject to prediction, for example a BTB or BTAC, then all branches are predictable branches.

If branches are decoded before the predictor, so that the branch prediction logic dynamically predicts only some branches, for example conditional and indirect branches, then it is IMPLEMENTATION DEFINED whether other branches are counted as predictable branches. Arm recommends that all branches are counted.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

An implementation might include other structures that predict branches, such as a loop buffer that predicts short backwards direct branches as taken. Each execution of such a branch is a predictable branch. Terminating the loop might generate a misprediction event that is counted by BR_MIS_PRED.

If no program-flow prediction resources are implemented, this event is optional, but Arm recommends that BR_PRED counts all branches.

It is IMPLEMENTATION DEFINED when the branch is counted. Arm recommends that it is counted when the branch is resolved, that is, at the same point in the instruction pipeline as when the BR_MIS_PRED event would be counted if the branch resolves as mispredicted. This means that (BR_PRED - BR_MIS_PRED) is the number of correctly predicted branches and the ratio (BR_MIS_PRED ÷ BR_PRED) can be calculated in a meaningful way.

PMCEID0_EL0[18] reads as 1 if this event is implemented and 0 otherwise.

`0x0013`**, MEM_ACCESS, Data memory access**

The counter counts each Memory-read operation or Memory-write operation that the PE makes.

The counter increments whether the access results in an access to a Level 1 data or unified cache, a Level 2 data or unified cache, or neither of these.

The number of accesses generated by each instruction is IMPLEMENTATION DEFINED.

The counter does not count:

- Instruction memory accesses.
- Translation table walks.
- Write-back from any cache.
- Refilling of any cache.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

PMCEID0_EL0[19] reads as 1 if this event is implemented and 0 otherwise.

`0x0014`**, L1I_CACHE, Level 1 instruction cache access**

The counter counts each Instruction memory access to at least the Level 1 instruction or unified cache.

Each Instruction memory access to other Level 1 instruction or unified memory structures, such as refill buffers, is also counted.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

If FEAT_PMUv3p4 is implemented, accesses that only update the cache status information for a cache entry without accessing the content of the cache entry are not counted.

If FEAT_PMUv3p4 is not implemented, it is IMPLEMENTATION DEFINED whether accesses that only update the cache status information for a cache entry without accessing the content of the cache entry are counted.

An example of cache status information is whether the cached data is held in an exclusive or shared state.

When the L1I_CACHE_RD event is implemented:

- If the L1I_CACHE_PRFM event is implemented, accesses to the Level 1 instruction cache due to a prefetch instruction are counted. Otherwise, these are not counted.
- If the L1I_CACHE_HWPRF event is implemented, accesses to the Level 1 instruction cache due to a hardware prefetcher are counted. Otherwise these events are not counted.

When the L1I_CACHE_RD event is not implemented, it IMPLEMENTATION DEFINED whether accesses to the Level 1 instruction cache due to prefetch instructions or due to a hardware prefetcher are counted.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID0_EL0[20] reads as 1 if this event is implemented and 0 otherwise.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x0015`**, L1D_CACHE_WB, Level 1 data cache write-back**

The counter counts each write-back of data from the Level 1 data or unified cache to outside of the Level 1 caches.

For example:

- A write-back of a dirty cache line to a Level 2 cache or memory.
- A write-back of a recently fetched cache line that has not been allocated to the Level 1 data cache.

Each write-back is counted once, even if multiple accesses are required to complete the write-back.

It is IMPLEMENTATION DEFINED whether the counter counts:

- A transfer of a dirty cache line from the Level 1 data cache to outside of Level 1 caches made as a result of a coherency request.
- Write-backs made as a result of cache maintenance instructions.

The counter does not count:

- The invalidation of a cache line without any write-back to outside of the Level 1 caches or memory.
- Writes that write through the Level 1 data cache to outside of the Level 1 caches.

A write-back is attributable to the agent that generated the request that caused the write-back. This might not be the same agent that caused the data being written back to be allocated into the cache.

An Unattributable write-back event occurs when a requestor outside of the PE makes a coherency request that results in write-back. If the cache is shared, then an Unattributable write-back event is not counted. If the cache is not shared, then the event is counted.

It is IMPLEMENTATION DEFINED whether a write of a whole cache line that is not the result of the eviction of a line from the cache, is counted. For example, this applies when the PE determines software is streaming writes to memory and does not allocate lines to the cache, or by a `DC ZVA` operation.

PMCEID0_EL0[21] reads as 1 if this event is implemented and 0 otherwise.

`0x0016`**, L2D_CACHE, Level 2 data cache access**

The counter counts each Memory-read operation or Memory-write operation that causes a cache access to at least the Level 2 data or unified cache.

Each access to a cache line is counted including refills of and write-backs from other caches. Each access to other Level 2 data or unified memory structures, for example refill buffers, write buffers, and write-back buffers, is also counted.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

If FEAT_PMUv3p4 is implemented, accesses that only update the cache status information for a cache entry without accessing the content of the cache entry are not counted.

If FEAT_PMUv3p4 is not implemented, it is IMPLEMENTATION DEFINED whether accesses that only update the cache status information for a cache entry without accessing the content of the cache entry are counted.

An example of cache status information is whether the cached data is held in an exclusive or shared state.

When the L2D_CACHE_RW event is implemented:

- If the L2D_CACHE_PRFM event is implemented, accesses to the Level 2 data cache due to a prefetch instruction are counted. Otherwise, these accesses are not counted.
- If the L2D_CACHE_HWPRF event is implemented, accesses to the Level 2 data cache due to a hardware prefetcher are counted. Otherwise these accesses are not counted.

When the L2D_CACHE_RW event is not implemented, it IMPLEMENTATION DEFINED whether accesses to the Level 2 data cache due to prefetch instructions or due to a hardware prefetcher are counted.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID0_EL0[22] reads as 1 if this event is implemented and 0 otherwise.

0x0017, L2D_CACHE_REFILL, Level 2 data cache refill

The counter counts each access counted by L2D_CACHE that causes a refill of the Level 2 data or unified cache, or any Level 1 data, instruction, or unified cache of this PE, from outside of those caches.

A refill includes any access that causes data to be fetched from outside of the Level 1 and Level 2 caches, even if the data is ultimately not allocated into the Level 2 data cache. For example, data might be fetched into a buffer but then discarded, rather than being allocated into a cache. These buffers are treated as part of the cache.

For example, the counter counts:

- Accesses to the Level 2 data cache that cause a refill that is satisfier by fetching data from memory, a Level 3 cache, or a Level 2 cache of another PE.
- Refills of and write-backs from any Level 1 data, instruction, or unified cache of this PE that cause a refill from outside of the Level 1 and Level 2 caches of this PE.
- Accesses to the Level 2 data cache that cause a refill of a Level 1 cache of this PE from outside of the Level 1 and Level 2 caches of this PE, even if there is no refill of the Level 2 data cache.

The counter does not count accesses that:

- Miss in the cache but are satisfied by the refill of a previous miss and do not cause a new refill, even if that previous refill is not complete at the time of the miss.
- Miss in the cache but do not generate a refill, such as a write through the cache.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID0_EL0[23] reads as 1 if this event is implemented and 0 otherwise.

0x0018, L2D_CACHE_WB, Level 2 data cache write-back

The counter counts each write-back of data from the Level 2 data or unified cache to outside of the Level 1 and Level 2 caches.

For example:

- A write-back of a dirty cache line to a Level 3 cache or memory.
- A write-back of a recently fetched cache line that has not been allocated to the Level 2 data cache.

Each write-back is counted once, even if multiple accesses are required to complete the write-back.

It is IMPLEMENTATION DEFINED whether the counter counts:

- A transfer of a dirty cache line from the Level 2 data cache to outside of Level 1 and Level 2 caches made as a result of a coherency request.
- Write-backs made as a result of cache maintenance instructions.

The counter does not count:

- The invalidation of a cache line without any write-back to outside of the Level 1 and Level 2 caches or memory.
- Writes that write through the Level 2 data cache to outside of the Level 1 and Level 2 caches.
- Transfers of data from the Level 2 data cache to a Level 1 cache to satisfy a refill of the other cache.

A write-back is attributable to the agent that generated the request that caused the write-back. This might not be the same agent that caused the data being written back to be allocated into the cache.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

An Unattributable write-back event occurs when a requestor outside of the PE makes a coherency request that results in write-back. If the cache is shared, then an Unattributable write-back event is not counted. If the cache is not shared, then the event is counted.

It is IMPLEMENTATION DEFINED whether a write of a whole cache line that is not the result of the eviction of a line from the cache, is counted. For example, this applies when the PE determines software is streaming writes to memory and does not allocate lines to the cache, or by a `DC ZVA` operation.

PMCEID0_EL0[24] reads as 1 if this event is implemented and 0 otherwise.

`0x0019`**, BUS_ACCESS, Bus access**

The counter counts each Memory-read operation or Memory-write operation that accesses outside of the boundary of the PE and its closely-coupled caches.

Where this boundary lies with respect to any implemented caches is IMPLEMENTATION DEFINED. Where an implementation has multiple buses at this boundary, this event counts the sum of accesses across all buses.

A bus access is part of a bus transaction. The exact nature of a bus transaction is IMPLEMENTATION DEFINED, but for the purposes of event monitoring consists of a single access comprising one or more cycles, or beats, when the transaction occupies the bus. The BUS_ACCESS event counts each beat of each transaction. That is, each bus cycle counted by BUS_CYCLES for which the bus is active.

Bus transactions include refills of and write-backs from data, instruction, and unified caches. Whether bus transactions include operations that use the bus but do not explicitly transfer data is IMPLEMENTATION DEFINED.

An Unattributable bus transaction occurs when a requestor outside the PE makes a request that results in a bus access, for example, a coherency request.

If a bus supports multiple accesses per cycle, for example through multiple channels, the counter increments once for each channel that is active on a cycle, and so it might increment by more than one in any given cycle.

The maximum increment in any given cycle is IMPLEMENTATION DEFINED.

PMCEID0_EL0[25] reads as 1 if this event is implemented and 0 otherwise.

`0x001A`**, MEMORY_ERROR, Local memory error**

The counter counts each occurrence of a memory error signaled by a memory closely coupled to this PE.

The definition of local memories is IMPLEMENTATION DEFINED but includes caches, tightly-coupled memories, and TLB arrays.

Memory error refers to a physical error detected by the hardware, such as a parity or ECC error. It includes errors that are correctable and those that are not. It does not include errors as defined in the architecture, such as MMU faults.

PMCEID0_EL0[26] reads as 1 if this event is implemented and 0 otherwise.

`0x001B`**, INST_SPEC, Operation speculatively executed**

The counter counts each speculatively executed instruction.

This includes instructions that are subsequently not architecturally executed.

PMCEID0_EL0[27] reads as 1 if this event is implemented and 0 otherwise.

`0x001D`**, BUS_CYCLES, Bus cycle**

The counter increments on every cycle of the interface at the boundary of the PE and its closely-coupled caches. Where this boundary lies with respect to any implemented caches is IMPLEMENTATION DEFINED.

Note

If the implementation clocks the external memory interface at the same rate as the processor hardware, then the counter counts every cycle.

PMCEID0_EL0[29] reads as 1 if this event is implemented and 0 otherwise.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x001F`, **L1D_CACHE_ALLOCATE, Level 1 data cache allocation without refill**

The counter counts each Memory-write operation that writes an entire line into the Level 1 data or unified cache without fetching data from outside the Level 1 data or unified cache.

These are allocations of cache lines in the Level 1 data or unified cache that are not refills counted by L1D_CACHE_REFILL. For example:

- A write of an entire cache line from a coalescing write buffer.
- An operation such as `DC ZVA`.

The counter counts only Memory-write operations Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, other PEs in the multithreaded implementation.

PMCEID0_EL0[31] reads as 1 if this event is implemented and 0 otherwise.

`0x0020`, **L2D_CACHE_ALLOCATE, Level 2 data cache allocation without refill**

The counter counts each Memory-write operation that writes an entire line into the Level 2 data or unified cache without fetching data from outside the Level 2 data or unified cache.

These are allocations of cache lines in the Level 2 data or unified cache that are not refills counted by L2D_CACHE_REFILL. For example:

- A write-back of an entire cache line from a Level 1 cache to the Level 2 data cache.
- A write of an entire cache line from a coalescing write buffer.
- An operation such as `DC ZVA`.

The counter counts only Memory-write operations Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, other PEs in the multithreaded implementation.

PMCEID1_EL0[0] reads as 1 if this event is implemented and 0 otherwise.

`0x0022`, **BR_MIS_PRED_RETIRED, Branch instruction architecturally executed, mispredicted**

The counter counts each branch instruction counted by BR_RETIRED that was not correctly predicted.

If no program-flow prediction resources are implemented, this event counts all retired not-taken branches.

PMCEID1_EL0[2] reads as 1 if this event is implemented and 0 otherwise.

`0x0023`, **STALL_FRONTEND, No operation sent for execution due to the frontend**

The counter counts each cycle counted by CPU_CYCLES where no Attributable instruction or operation was sent for execution and there was no Attributable instruction or operation available to dispatch for the PE from the frontend.

The division between frontend and backend is IMPLEMENTATION DEFINED. All STALL events are counted at the same point in the pipeline.

Note

- For a simplified pipeline model of Fetch-Decode-Issue-Execute-Retire, Arm recommends that the events are counted when instructions are dispatched from Decode to Issue.
- In a single cycle, both the STALL_BACKEND and STALL_FRONTEND events might be counted, if both the backend is unable to accept any operations and there are no operations available to issue from the frontend.

PMCEID1_EL0[3] reads as 1 if this event is implemented and 0 otherwise.

`0x0024`, **STALL_BACKEND, No operation sent for execution due to the backend**

The counter counts each cycle counted by CPU_CYCLES where Attributable instructions or operations for the PE are available to dispatch from the frontend to the backend, but no Attributable instruction or operation is sent for execution because the backend is unable to accept any of the instructions or operations available for the PE.

For example, the backend might be unable to accept operations because of a resource conflict or non-availability.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

It is IMPLEMENTATION DEFINED whether the counter also counts each cycle counted by CPU_CYCLES where no Attributable instructions or operations for the PE are available to dispatch from the frontend and the backend is unable to accept any instructions or operations for the PE.

The division between frontend and backend is IMPLEMENTATION DEFINED. For more information, see STALL_FRONTEND. All STALL events are counted at the same point in the pipeline.

Note

In a single cycle, both the STALL_BACKEND and STALL_FRONTEND events might be counted, if both the backend is unable to accept any operations and there are no operations available to issue from the frontend.

PMCEID1_EL0[4] reads as 1 if this event is implemented and 0 otherwise.

`0x0025`**, L1D_TLB, Level 1 data TLB access**

The counter counts each Memory-read operation or Memory-write operation that causes a TLB access to at least the Level 1 data or unified TLB.

Each access to a TLB entry is counted including multiple accesses caused by single instructions such as LDM or STM.

The counter does not count the access if the access is due to a TLB maintenance instruction.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID1_EL0[5] reads as 1 if this event is implemented and 0 otherwise.

`0x0026`**, L1I_TLB, Level 1 instruction TLB access**

The counter counts each Instruction memory access that causes a TLB access to at least the Level 1 instruction TLB.

The counter does not count the access if the access is due to a TLB maintenance instruction.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID1_EL0[6] reads as 1 if this event is implemented and 0 otherwise.

`0x0027`**, L2I_CACHE, Level 2 instruction cache access**

The counter counts each Instruction memory access to at least the Level 2 instruction or unified cache.

Each Instruction memory access to other Level 2 instruction or unified memory structures, such as refill buffers, is also counted.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

If FEAT_PMUv3p4 is implemented, accesses that only update the cache status information for a cache entry without accessing the content of the cache entry are not counted.

If FEAT_PMUv3p4 is not implemented, it is IMPLEMENTATION DEFINED whether accesses that only update the cache status information for a cache entry without accessing the content of the cache entry are counted.

An example of cache status information is whether the cached data is held in an exclusive or shared state.

When the L2I_CACHE_RD event is implemented:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the L2I_CACHE_PRFM event is implemented, accesses to the Level 2 instruction cache due to a prefetch instruction are counted. Otherwise, these are not counted.
- If the L2I_CACHE_HWPRF event is implemented, accesses to the Level 2 instruction cache due to a hardware prefetcher are counted. Otherwise these events are not counted.

When the L2I_CACHE_RD event is not implemented, it IMPLEMENTATION DEFINED whether accesses to the Level 2 instruction cache due to prefetch instructions or due to a hardware prefetcher are counted.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID1_EL0[7] reads as 1 if this event is implemented and 0 otherwise.

`0x0028`**, L2I_CACHE_REFILL, Level 2 instruction cache refill**

The counter counts each access counted by L2I_CACHE that causes a refill of the Level 2 instruction or unified cache, or any Level 1 data, instruction, or unified cache of this PE, from outside of those caches.

A refill includes any access that causes data to be fetched from outside of the Level 1 and Level 2 caches, even if the data is ultimately not allocated into the Level 2 instruction cache. For example, data might be fetched into a buffer but then discarded, rather than being allocated into a cache. These buffers are treated as part of the cache.

For example, the counter counts:

- Accesses to the Level 2 instruction cache that cause a refill that is satisfied by fetching data from memory, a Level 3 cache, or a Level 2 cache of another PE.
- Refills of any Level 1 instruction or unified cache of this PE that cause a refill from outside of the Level 1 and Level 2 caches of this PE.
- Accesses to the Level 2 instruction cache that cause a refill of a Level 1 cache of this PE from outside of the Level 1 and Level 2 caches of this PE, even if there is no refill of the Level 2 instruction cache.

The counter does not count accesses that miss in the cache but are satisfied by the refill of a previous miss and do not cause a new refill, even if that previous refill is not complete at the time of the miss.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID1_EL0[8] reads as 1 if this event is implemented and 0 otherwise.

`0x0029`**, L3D_CACHE_ALLOCATE, Level 3 data cache allocation without refill**

The counter counts each Memory-write operation that writes an entire line into the Level 3 data or unified cache without fetching data from outside the Level 3 data or unified cache.

These are allocations of cache lines in the Level 3 data or unified cache that are not refills counted by L3D_CACHE_REFILL. For example:

- A write-back of an entire cache line from a Level 2 cache to the Level 3 data cache.
- A write of an entire cache line from a coalescing write buffer.
- An operation such as `DC ZVA`.

The counter counts only Memory-write operations Attributable to the PE counting the event, and, if the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, other PEs in the multithreaded implementation.

PMCEID1_EL0[9] reads as 1 if this event is implemented and 0 otherwise.

`0x002A`**, L3D_CACHE_REFILL, Level 3 data cache refill**

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The counter counts each access counted by L3D_CACHE that causes a refill of the Level 3 data or unified cache, or any Level 1 or Level 2 data, instruction, or unified cache of this PE, from outside of those caches.

A refill includes any access that causes data to be fetched from outside of the Level 1 to Level 3 caches, even if the data is ultimately not allocated into the Level 3 data cache. For example, data might be fetched into a buffer but then discarded, rather than being allocated into a cache. These buffers are treated as part of the cache.

For example, the counter counts:

- Accesses to the Level 3 data cache that cause a refill that is satisfier by fetching data from memory, a lower level cache, or a Level 3 cache of another PE.
- Refills of and write-backs from any Level 1 or Level 2 data, instruction, or unified caches of this PE that cause a refill from outside of the Level 1 to Level 3 caches of this PE.
- Accesses to the Level 3 data cache that cause a refill of a Level 2 cache of this PE from outside of the Level 1 to Level 3 caches of this PE, even if there is no refill of the Level 3 data cache.

The counter does not count accesses that:

- Miss in the cache but are satisfied by the refill of a previous miss and do not cause a new refill, even if that previous refill is not complete at the time of the miss.
- Miss in the cache but do not generate a refill, such as a write through the cache.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID1_EL0[10] reads as 1 if this event is implemented and 0 otherwise.

`0x002B`**, L3D_CACHE, Level 3 data cache access**

The counter counts each Memory-read operation or Memory-write operation that causes a cache access to at least the Level 3 data or unified cache.

Each access to a cache line is counted including refills of and write-backs from other caches. Each access to other Level 3 data or unified memory structures, for example refill buffers, write buffers, and write-back buffers, is also counted.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

If FEAT_PMUv3p4 is implemented, accesses that only update the cache status information for a cache entry without accessing the content of the cache entry are not counted.

If FEAT_PMUv3p4 is not implemented, it is IMPLEMENTATION DEFINED whether accesses that only update the cache status information for a cache entry without accessing the content of the cache entry are counted.

An example of cache status information is whether the cached data is held in an exclusive or shared state.

When the L3D_CACHE_RW event is implemented:

- If the L3D_CACHE_PRFM event is implemented, accesses to the Level 3 data cache due to a prefetch instruction are counted. Otherwise, these accesses are not counted.
- If the L3D_CACHE_HWPRF event is implemented, accesses to the Level 3 data cache due to a hardware prefetcher are counted. Otherwise these accesses are not counted.

When the L3D_CACHE_RW event is not implemented, it IMPLEMENTATION DEFINED whether accesses to the Level 3 data cache due to prefetch instructions or due to a hardware prefetcher are counted.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PMCEID1_EL0[11] reads as 1 if this event is implemented and 0 otherwise.

`0x002C`, **L3D_CACHE_WB, Level 3 data cache write-back**

The counter counts each write-back of data from the Level 3 data or unified cache to outside of the Level 1 to Level 3 caches.

For example:

- A write-back of a dirty cache line to a lower level cache or memory.
- A write-back of a recently fetched cache line that has not been allocated to the Level 3 data cache.

Each write-back is counted once, even if multiple accesses are required to complete the write-back.

It is IMPLEMENTATION DEFINED whether the counter counts:

- A transfer of a dirty cache line from the Level 3 data cache to outside of Level 1 to Level 3 caches made as a result of a coherency request.
- Write-backs made as a result of cache maintenance instructions.

The counter does not count:

- The invalidation of a cache line without any write-back to outside of the Level 1 to Level 3 caches or memory.
- Writes that write through the Level 3 data cache to outside of the Level 1 to Level 3 caches.
- Transfers of data from the Level 3 data cache to a Level 1 or Level 2 cache to satisfy a refill of the other cache.

A write-back is attributable to the agent that generated the request that caused the write-back. This might not be the same agent that caused the data being written back to be allocated into the cache.

An Unattributable write-back event occurs when a requestor outside of the PE makes a coherency request that results in write-back. If the cache is shared, then an Unattributable write-back event is not counted. If the cache is not shared, then the event is counted.

It is IMPLEMENTATION DEFINED whether a write of a whole cache line that is not the result of the eviction of a line from the cache, is counted. For example, this applies when the PE determines software is streaming writes to memory and does not allocate lines to the cache, or by a `DC ZVA` operation.

PMCEID1_EL0[12] reads as 1 if this event is implemented and 0 otherwise.

`0x002D`, **L2D_TLB_REFILL, Level 2 data TLB refill**

The counter counts each access counted by L2D_TLB that causes a TLB refill of the Level 2 data or unified TLB, or any Level 1 data, instruction, or unified TLBs of this PE.

This includes each access that causes a memory access due to a translation table walk or an access to another level of TLB caching.

The counter does not count the access if any of the following are true:

- The access is due to a TLB maintenance instruction.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the access if any of the following are true:

- The access misses in the TLB and generates a translation table walk, but the result is not allocated into the TLB for any reason other than a Translation fault.
- The access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PMCEID1_EL0[13] reads as 1 if this event is implemented and 0 otherwise.

`0x002E`, **L2I_TLB_REFILL, Level 2 instruction TLB refill**

The counter counts each Instruction memory access counted by L2I_TLB that causes a TLB refill of the Level 2 instruction TLB, or any Level 1 data, instruction, or unified TLBs of this PE.

This includes each Instruction memory access that causes a memory access due to a translation table walk or an access to another level of TLB caching.

The counter does not count the access if any of the following are true:

- The access is due to a TLB maintenance instruction.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the access if any of the following are true:

- The access misses in the TLB and generates a translation table walk, but the result is not allocated into the TLB for any reason other than a Translation fault.
- The access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID1_EL0[14] reads as 1 if this event is implemented and 0 otherwise.

`0x002F`, **L2D_TLB, Level 2 data TLB access**

The counter counts each Memory-read operation or Memory-write operation that causes a TLB access to at least the Level 2 data or unified TLB.

Each access to a TLB entry is counted including refills of Level 1 TLBs.

The counter does not count the access if the access is due to a TLB maintenance instruction.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID1_EL0[15] reads as 1 if this event is implemented and 0 otherwise.

`0x0030`, **L2I_TLB, Level 2 instruction TLB access**

The counter counts each Instruction memory access that causes a TLB access to at least the Level 2 instruction TLB.

The counter does not count the access if the access is due to a TLB maintenance instruction.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID1_EL0[16] reads as 1 if this event is implemented and 0 otherwise.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x0031`, **REMOTE_ACCESS, Access to a remote device**

The counter counts each Memory-read operation or Memory-write operation that causes an access to a remote device.

That is, a socket that does not contain the PE.

System topology is IMPLEMENTATION DEFINED. This means that it is IMPLEMENTATION DEFINED which systems are defined as multi-socket systems, and, in systems defined as multi-socket systems, which components are defined as being in the same or different sockets. Arm recommends that devices where an access through that device incurs a significant latency penalty compared to other accesses are treated as being in a different socket.

For example, in a system comprising multiple integrated circuits in a multi-chip module, an access to a different integrated circuit in the same module might be treated as an access to another socket, even though the multi-chip module is physically connected to a single socket at the motherboard. However, in another system with many such multi-chip modules, an access to a different integrated system in the same module might be treated as an access to the same socket because an access to an integrated circuit on a different module has much higher latency.

It is IMPLEMENTATION DEFINED whether an access that causes a snoop into a different socket but does not return data from or pass data to the remote socket is counted.

PMCEID1_EL0[17] reads as 1 if this event is implemented and 0 otherwise.

`0x0032`, **LL_CACHE, Last level cache access**

The counter counts each Memory-read operation or Memory-write operation that causes a cache access to at least the Last level cache.

Each access to a cache line is counted including refills of and write-backs from other caches. Each access to other Last level data or unified memory structures, for example refill buffers, write buffers, and write-back buffers, is also counted.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

If FEAT_PMUv3p4 is implemented, accesses that only update the cache status information for a cache entry without accessing the content of the cache entry are not counted.

If FEAT_PMUv3p4 is not implemented, it is IMPLEMENTATION DEFINED whether accesses that only update the cache status information for a cache entry without accessing the content of the cache entry are counted.

An example of cache status information is whether the cached data is held in an exclusive or shared state.

When the LL_CACHE_RW event is implemented:

- If the LL_CACHE_PRFM event is implemented, accesses to the Last level cache due to a prefetch instruction are counted. Otherwise, these accesses are not counted.
- If the LL_CACHE_HWPRF event is implemented, accesses to the Last level cache due to a hardware prefetcher are counted. Otherwise these accesses are not counted.

When the LL_CACHE_RW event is not implemented, it IMPLEMENTATION DEFINED whether accesses to the Last level cache due to prefetch instructions or due to a hardware prefetcher are counted.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID1_EL0[18] reads as 1 if this event is implemented and 0 otherwise.

`0x0033`, **LL_CACHE_MISS, Last level cache miss**

If the LL_CACHE_RW event is implemented, the counter counts each access counted by LL_CACHE_RW that is not completed by the Last level cache.

If the LL_CACHE_RW event is not implemented, the counter counts each access counted by LL_CACHE that is not completed by the Last level cache.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

That is, the access is one of the following:

- A Memory-read operation that does not return data from the Last level cache.
- A Memory-write operation that does not update the Last level cache.

The counter does not count operations that are completed by a cache above the Last level cache.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID1_EL0[19] reads as 1 if this event is implemented and 0 otherwise.

`0x0034`**, DTLB_WALK, Data TLB access with at least one translation table walk**

The counter counts each access counted by L1D_TLB that causes a refill of a data or unified TLB involving at least one translation table walk access.

This includes each complete or partial translation table walk that causes an access to memory, including to data or translation table walk caches.

It is IMPLEMENTATION DEFINED whether accesses that cause an update of an existing TLB entry involving at least one translation table walk access are counted.

The counter does not count the access if any of the following are true:

- The access is due to a TLB maintenance instruction.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the access if the access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID1_EL0[20] reads as 1 if this event is implemented and 0 otherwise.

`0x0035`**, ITLB_WALK, Instruction TLB access with at least one translation table walk**

The counter counts each access counted by L1I_TLB that causes a refill of an instruction TLB involving at least one translation table walk access.

This includes each complete or partial translation table walk that causes an access to memory, including to data or translation table walk caches.

It is IMPLEMENTATION DEFINED whether accesses that cause an update of an existing TLB entry involving at least one translation table walk access are counted.

The counter does not count the access if any of the following are true:

- The access is due to a TLB maintenance instruction.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the access if the access generates a Translation fault for any other reason.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID1_EL0[21] reads as 1 if this event is implemented and 0 otherwise.

`0x0036`**, LL_CACHE_RD, Last level cache access, read**

If the LL_CACHE_RW event is implemented, the counter counts each access counted by LL_CACHE_RW that is a Memory-read operation.

If the LL_CACHE_RW event is not implemented, the counter counts each access counted by LL_CACHE that is a Memory-read operation.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID1_EL0[22] reads as 1 if this event is implemented and 0 otherwise.

`0x0037`**, LL_CACHE_MISS_RD, Last level cache miss, read**

The counter counts each access counted by both LL_CACHE_MISS and LL_CACHE_RD.

That is, every miss of the Last level cache counted by LL_CACHE_MISS that is caused by a Memory-read operation.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID1_EL0[23] reads as 1 if this event is implemented and 0 otherwise.

`0x0038`**, REMOTE_ACCESS_RD, Access to a remote device, read**

The counter counts each access counted by REMOTE_ACCESS that is a Memory-read operation.

PMCEID1_EL0[24] reads as 1 if this event is implemented and 0 otherwise.

`0x0039`**, L1D_CACHE_LMISS_RD, Level 1 data cache long-latency read miss**

The counter counts each Memory-read operation to the Level 1 data or unified cache counted by L1D_CACHE that incurs additional latency because it returns data from outside of the Level 1 data or unified cache of this PE.

The event indicates to software that the access missed in the Level 1 data or unified cache and might have a significant performance impact due to the additional latency compared to the latency of an access that hits in the Level 1 data or unified cache.

The counter does not count:

- Accesses where the additional latency is unlikely to be significantly performance-impacting. For example, if the access hits in another cache in the same local cluster, and the additional latency is small when compared to a miss in all Level 1 caches that the access looks up in and results in an access being made to a Level 2 cache or elsewhere beyond the Level 1 data or unified cache.
- A miss that does not cause a new cache refill but is satisfied from a previous miss.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

An implementation is not required to measure the latency, nor to track the access to determine whether the additional latency caused a performance impact. An implementation can extend the definition of this event with additional scenarios where an access might have a significant performance impact due to additional latency for the access.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance operations are counted.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID1_EL0[25] reads as 1 if this event is implemented and 0 otherwise.

`0x003A`, **OP_RETIRED, Micro-operation architecturally executed**

The counter counts each operation counted by OP_SPEC that would be executed in a Simple sequential execution of the program.

PMCEID1_EL0[26] reads as 1 if this event is implemented and 0 otherwise.

`0x003B`, **OP_SPEC, Micro-operation speculatively executed**

The counter counts each operation speculatively executed executed by the PE.

This includes operations that would not be executed in a Simple sequential execution of the program.

PMCEID1_EL0[27] reads as 1 if this event is implemented and 0 otherwise.

`0x003C`, **STALL, No operation sent for execution**

The counter counts each cycle counted by CPU_CYCLES where no operation was sent for execution.

On a multithreaded implementation:

- If the Effective value of PMEVTYPER<n>_EL0.MT is 0, then the counter counts cycles when the PE is active and no operation for the PE is sent for execution, even if operations Attributable to other PEs in the multithreaded implementation are sent for execution in that cycle. The counter does not count cycles when the PE is not active.
- If the Effective value of PMEVTYPER<n>_EL0.MT is 1, then the counter counts all cycles when no instructions or operations for any PE in the multithreaded implementation were sent for execution.

All STALL events are counted at the same point in the pipeline.

PMCEID1_EL0[28] reads as 1 if this event is implemented and 0 otherwise.

`0x003D`, **STALL_SLOT_BACKEND, No operation sent for execution on a Slot due to the backend**

The counter counts each Slot counted by STALL_SLOT where an Attributable instruction or operation for the PE is available on the Slot to dispatch from the frontend to the backend, but is not sent for execution because the backend is unable to accept the available instruction or operation for the PE.

It is IMPLEMENTATION DEFINED whether the counter also counts each Slot counted by STALL_SLOT where no Attributable instruction or operation for the PE is available on the Slot to dispatch from the frontend and the backend is unable to accept any instruction or operation for the PE.

The division between frontend and backend is IMPLEMENTATION DEFINED. For more information, see STALL_FRONTEND. All STALL events are counted at the same point in the pipeline. The maximum value by which STALL_SLOT_BACKEND can count in a single cycle is an IMPLEMENTATION DEFINED fixed value, `slots`.For more information, see STALL_SLOT.

Note

In a single cycle, the sum of values counted by STALL_SLOT_BACKEND and STALL_SLOT_FRONTEND might be greater-than `slots`, if both the backend is unable to accept some operations and there are fewer-than `slots` operations available to dispatch from the frontend.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PMCEID1_EL0[29] reads as 1 if this event is implemented and 0 otherwise.

`0x003E`, **STALL_SLOT_FRONTEND, No operation sent for execution on a Slot due to the frontend**

The counter counts each Slot counted by STALL_SLOT where no Attributable instruction or operation was sent for execution and there was no Attributable instruction or operation available to dispatch for the PE from the frontend for the Slot.

The division between frontend and backend is IMPLEMENTATION DEFINED. For more information, see STALL_FRONTEND. All STALL events are counted at the same point in the pipeline. The maximum value by which STALL_SLOT_FRONTEND can count in a single cycle is an IMPLEMENTATION DEFINED fixed value, `slots`.For more information, see STALL_SLOT.

Note

In a single cycle, the sum of values counted by STALL_SLOT_BACKEND and STALL_SLOT_FRONTEND might be greater-than `slots`, if both the backend is unable to accept some operations and there are fewer-than `slots` operations available to dispatch from the frontend.

PMCEID1_EL0[30] reads as 1 if this event is implemented and 0 otherwise.

`0x003F`, **STALL_SLOT, No operation sent for execution on a Slot**

The counter increments by the number of instruction or operation Slots that were not occupied by an instruction or operation Attributable to the PE on each cycle.

All STALL events are counted at the same point in the pipeline. The maximum value by which STALL_SLOT can count in a single cycle is an IMPLEMENTATION DEFINED fixed value, `slots`. The definition of a Slot is IMPLEMENTATION DEFINED. The formula STALL_SLOT÷(CPU_CYCLES×`slots`) gives the utilization of the Slots of the processor by Attributable instruction or operations of this PE. Each Slot holds at most one instruction or operation each cycle.

On a multithreaded implementation:

- If the Effective value of PMEVTYPER<n>_EL0.MT is 0, then the counter counts Slots occupied by an instruction or operation Attributable to other PEs of the multithreaded implementation only when the PE was active in that cycle. The counter does not count Slots on cycles when the PE was not active.
- If the Effective value of PMEVTYPER<n>_EL0.MT is 1, then, for every cycle, the counter counts all Slots not occupied by any instruction or operation for any PE of the multithreaded implementation.

PMCEID1_EL0[31] reads as 1 if this event is implemented and 0 otherwise.

`0x0040`, **L1D_CACHE_RD, Level 1 data cache access, read**

If the L1D_CACHE_RW event is implemented, the counter counts each access counted by L1D_CACHE_RW that is a Memory-read operation.

If the L1D_CACHE_RW event is not implemented, the counter counts each access counted by L1D_CACHE that is a Memory-read operation.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x0041`, **L1D_CACHE_WR, Level 1 data cache access, write**

If the L1D_CACHE_RW event is implemented, the counter counts each access counted by L1D_CACHE_RW that is a Memory-write operation.

If the L1D_CACHE_RW event is not implemented, the counter counts each access counted by L1D_CACHE that is a Memory-write operation.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x0042`**, L1D_CACHE_REFILL_RD, Level 1 data cache refill, read**

The counter counts each access counted by both L1D_CACHE_RD and L1D_CACHE_REFILL.

That is, every refill of the Level 1 data or unified cache counted by L1D_CACHE_REFILL that is caused by a Memory-read operation.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x0043`**, L1D_CACHE_REFILL_WR, Level 1 data cache refill, write**

The counter counts each access counted by both L1D_CACHE_REFILL and L1D_CACHE_WR.

That is, every refill of the Level 1 data or unified cache counted by L1D_CACHE_REFILL that is caused by a Memory-write operation.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x0044`**, L1D_CACHE_REFILL_INNER, Level 1 data cache refill, inner**

The counter counts each access counted by L1D_CACHE_REFILL that generates a refill satisfied by transfer from another cache inside of the immediate cluster.

Note

The boundary between inner and outer is IMPLEMENTATION DEFINED, and it is not necessarily linked to other similar boundaries, such as the boundary between Inner Cacheable and Outer Cacheable or the boundary between Inner Shareable and Outer Shareable.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x0045`**, L1D_CACHE_REFILL_OUTER, Level 1 data cache refill, outer**

The counter counts each access counted by L1D_CACHE_REFILL that generates a refill satisfied by transfer from outside of the immediate cluster.

Note

The boundary between inner and outer is IMPLEMENTATION DEFINED, and it is not necessarily linked to other similar boundaries, such as the boundary between Inner Cacheable and Outer Cacheable or the boundary between Inner Shareable and Outer Shareable.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x0046`**, L1D_CACHE_WB_VICTIM, Level 1 data cache write-back, victim**

The counter counts each write-back counted by L1D_CACHE_WB that occurs because the line is allocated for an access made by the PE.

`0x0047`**, L1D_CACHE_WB_CLEAN, Level 1 data cache write-back, cleaning and coherency**

The counter counts each write-back counted by L1D_CACHE_WB that occurs because of a coherency operation made by another PE or, optionally, the execution of a cache maintenance instruction.

Whether write-backs that are caused by the execution of a cache maintenance instruction are counted is IMPLEMENTATION DEFINED.

Note

The transfer of a dirty cache line from the Level 1 data cache of this PE to the data cache of another PE due to a hardware coherency operation is not counted unless the dirty cache line is also written back to a Level 2 cache or memory.

If a coherency request from a requestor outside of the PE results in a write-back, it is an Unattributable event.

`0x0048`**, L1D_CACHE_INVAL, Level 1 data cache invalidate**

The counter counts each invalidation of a cache line in the Level 1 data or unified cache. For example:

- Invalidation of a cache line because of a cache maintenance operation.
- Transfer of ownership of a cache line to another cache because of a coherency or refill request.

The counter does not count events if a cache refill of the Level 1 data or unified cache invalidates a line in the Level 1 data or unified cache.

If FEAT_PMUv3p4 is not implemented, the counter does not count locally-executed cache maintenance instructions that operate by set/way.

If FEAT_PMUv3p4 is implemented, it is IMPLEMENTATION DEFINED whether the counter counts locally-executed cache maintenance instructions that operate by set/way.

Note

Software that uses this event must know whether the Level 1 data cache is shared with other PEs. This event does not follow the general rule of Level 1 data cache events of only counting Attributable events.

`0x004C`**, L1D_TLB_REFILL_RD, Level 1 data TLB refill, read**

The counter counts each access counted by both L1D_TLB_RD and L1D_TLB_REFILL.

The counter does not count the access if any of the following are true:

- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the access if the access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x004D`, **L1D_TLB_REFILL_WR, Level 1 data TLB refill, write**

The counter counts each access counted by both L1D_TLB_REFILL and L1D_TLB_WR.

The counter does not count the access if any of the following are true:

- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the access if the access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x004E`, **L1D_TLB_RD, Level 1 data TLB access, read**

If the L1D_TLB_RW event is implemented, the counter counts each access counted by L1D_TLB_RW that is a Memory-read operation.

If the L1D_TLB_RW event is not implemented, the counter counts each access counted by L1D_TLB that is a Memory-read operation.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x004F`, **L1D_TLB_WR, Level 1 data TLB access, write**

If the L1D_TLB_RW event is implemented, the counter counts each access counted by L1D_TLB_RW that is a Memory-write operation.

If the L1D_TLB_RW event is not implemented, the counter counts each access counted by L1D_TLB that is a Memory-write operation.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x0050`, **L2D_CACHE_RD, Level 2 data cache access, read**

If the L2D_CACHE_RW event is implemented, the counter counts each access counted by L2D_CACHE_RW that is a Memory-read operation.

If the L2D_CACHE_RW event is not implemented, the counter counts each access counted by L2D_CACHE that is a Memory-read operation.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x0051`**, L2D_CACHE_WR, Level 2 data cache access, write**

If the L2D_CACHE_RW event is implemented, the counter counts each access counted by L2D_CACHE_RW that is a Memory-write operation.

If the L2D_CACHE_RW event is not implemented, the counter counts each access counted by L2D_CACHE that is a Memory-write operation.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x0052`**, L2D_CACHE_REFILL_RD, Level 2 data cache refill, read**

The counter counts each access counted by both L2D_CACHE_RD and L2D_CACHE_REFILL.

That is, every refill of the Level 2 data or unified cache counted by L2D_CACHE_REFILL that is caused by a Memory-read operation.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x0053`**, L2D_CACHE_REFILL_WR, Level 2 data cache refill, write**

The counter counts each access counted by both L2D_CACHE_REFILL and L2D_CACHE_WR.

That is, every refill of the Level 2 data or unified cache counted by L2D_CACHE_REFILL that is caused by a Memory-write operation.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x0056`**, L2D_CACHE_WB_VICTIM, Level 2 data cache write-back, victim**

The counter counts each write-back counted by L2D_CACHE_WB that occurs because the line is allocated for an access made by the PE.

`0x0057`**, L2D_CACHE_WB_CLEAN, Level 2 data cache write-back, cleaning and coherency**

The counter counts each write-back counted by L2D_CACHE_WB that occurs because of a coherency operation made by another PE or, optionally, the execution of a cache maintenance instruction.

Whether write-backs that are caused by the execution of a cache maintenance instruction are counted is IMPLEMENTATION DEFINED.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

The transfer of a dirty cache line from the Level 2 data cache of this PE to the data cache of another PE due to a hardware coherency operation is not counted unless the dirty cache line is also written back to a Level 3 cache or memory.

If a coherency request from a requestor outside of the PE results in a write-back, it is an Unattributable event.

`0x0058`**, L2D_CACHE_INVAL, Level 2 data cache invalidate**

The counter counts each invalidation of a cache line in the Level 2 data or unified cache. For example:

- Invalidation of a cache line because of a cache maintenance operation.
- Transfer of ownership of a cache line to another cache because of a coherency or refill request.

The counter does not count events if a cache refill of the Level 2 data or unified cache invalidates a line in the Level 2 data or unified cache.

If FEAT_PMUv3p4 is not implemented, the counter does not count locally-executed cache maintenance instructions that operate by set/way.

If FEAT_PMUv3p4 is implemented, it is IMPLEMENTATION DEFINED whether the counter counts locally-executed cache maintenance instructions that operate by set/way.

Note

Software that uses this event must know whether the Level 2 data cache is shared with other PEs. This event does not follow the general rule of Level 2 data cache events of only counting Attributable events.

`0x005C`**, L2D_TLB_REFILL_RD, Level 2 data TLB refill, read**

The counter counts each access counted by both L2D_TLB_RD and L2D_TLB_REFILL.

The counter does not count the access if any of the following are true:

- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the access if the access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x005D`**, L2D_TLB_REFILL_WR, Level 2 data TLB refill, write**

The counter counts each access counted by both L2D_TLB_REFILL and L2D_TLB_WR.

The counter does not count the access if any of the following are true:

- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the access if the access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x005E`, **L2D_TLB_RD, Level 2 data TLB access, read**

The counter counts each access counted by L2D_TLB that is a Memory-read operation.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x005F`, **L2D_TLB_WR, Level 2 data TLB access, write**

The counter counts each access counted by L2D_TLB that is a Memory-write operation.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x0060`, **BUS_ACCESS_RD, Bus access, read**

The counter counts each access counted by BUS_ACCESS that is a Memory-read operation.

`0x0061`, **BUS_ACCESS_WR, Bus access, write**

The counter counts each access counted by BUS_ACCESS that is a Memory-write operation.

`0x0062`, **BUS_ACCESS_SHARED, Bus access, Normal, Cacheable, Shareable**

The counter counts each access counted by BUS_ACCESS that is Normal, Cacheable, Shareable.

Note

It is IMPLEMENTATION DEFINED how the PE translates the attributes from the translation table entry for a region to the attributes on the bus.
In particular, a region of memory designated as Normal, Cacheable, Inner Shareable, Not Outer Shareable by a translation table entry, might be marked as either shareable or Non-shareable at the boundary of the PE and its closely-coupled caches. This depends on where the IMPLEMENTATION DEFINED boundary lies, between Inner and Outer Shareable.
If the Inner Shareable extends beyond the PE boundary, and the bus indicates the distinction between Inner and Outer Shareable, then either is counted as shareable for the purposes of defining this event.

`0x0063`, **BUS_ACCESS_NOT_SHARED, Bus access, not Normal, Cacheable, Shareable**

The counter counts each access counted by BUS_ACCESS that is not counted by BUS_ACCESS_SHARED.

For example, the counter counts accesses marked as:

- Normal, Cacheable, Non-shareable.
- Normal, Non-cacheable.
- Device.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

It is IMPLEMENTATION DEFINED, how the PE translates the attributes from the translation table entries for a region to the attributes on the bus.
In particular, a region of memory designated as Normal, Cacheable, Inner Shareable, Not Outer Shareable by a translation table entry, might be marked as either shareable or Non-shareable at the boundary of the PE and its closely-coupled caches. This depends on where the IMPLEMENTATION DEFINED boundary lies, between Inner and Outer Shareable.
If the Inner Shareable extends beyond the PE boundary, and the bus indicates the distinction between Inner and Outer Shareable, then either is counted as shareable for the purposes of defining this event.

`0x0064`**, BUS_ACCESS_NORMAL, Bus access, normal**

The counter counts each access counted by BUS_ACCESS that is to Normal or bulk memory.

For example, the counter counts Normal, Cacheable and Normal, Non-cacheable accesses but does not count Device accesses.

`0x0065`**, BUS_ACCESS_PERIPH, Bus access, peripheral**

The counter counts each access counted by BUS_ACCESS that is not counted by BUS_ACCESS_NORMAL.

For example, the counter counts accesses to Device memory.

`0x0066`**, MEM_ACCESS_RD, Data memory access, read**

If the MEM_ACCESS_RW event is implemented, the counter counts each access counted by MEM_ACCESS_RW that is a Memory-read operation.

If the MEM_ACCESS_RW event is not implemented, the counter counts each access counted by MEM_ACCESS that is a Memory-read operation.

`0x0067`**, MEM_ACCESS_WR, Data memory access, write**

If the MEM_ACCESS_RW event is implemented, the counter counts each access counted by MEM_ACCESS_RW that is a Memory-write operation.

If the MEM_ACCESS_RW event is not implemented, the counter counts each access counted by MEM_ACCESS that is a Memory-write operation.

`0x0068`**, UNALIGNED_LD_SPEC, Unaligned access, read**

The counter counts each unaligned access counted by UNALIGNED_LDST_SPEC that is a Memory-read operation.

The unaligned access is counted even if it is subsequently transformed into multiple aligned accesses.

`0x0069`**, UNALIGNED_ST_SPEC, Unaligned access, write**

The counter counts each unaligned access counted by UNALIGNED_LDST_SPEC that is a Memory-write operation.

The unaligned access is counted even if it is subsequently transformed into multiple aligned accesses.

`0x006A`**, UNALIGNED_LDST_SPEC, Unaligned access**

The counter counts each access counted by MEM_ACCESS that is an unaligned Memory-read operation or unaligned Memory-write operation.

The unaligned access is counted even if it is subsequently transformed into multiple aligned accesses.

`0x006C`**, LDREX_SPEC, Exclusive operation speculatively executed, Load-Exclusive**

The counter counts each speculatively executed Load-Exclusive instruction.

The definition of speculatively executed is IMPLEMENTATION DEFINED but must be the same as for the STREX_SPEC event.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x006D`, **STREX_PASS_SPEC, Exclusive operation speculatively executed, Store-Exclusive pass**

The counter counts each speculatively executed Store-Exclusive instruction counted by STREX_SPEC that completed a write.

`0x006E`, **STREX_FAIL_SPEC, Exclusive operation speculatively executed, Store-Exclusive fail**

The counter counts each speculatively executed Store-Exclusive instruction counted by STREX_SPEC that fails to complete a write.

It is within the IMPLEMENTATION DEFINED definition of speculatively executed whether this includes conditional instructions that fail the condition code check.

`0x006F`, **STREX_SPEC, Exclusive operation speculatively executed, Store-Exclusive**

The counter counts each speculatively executed Store-Exclusive instruction.

The definition of speculatively executed is IMPLEMENTATION DEFINED but it must be the same as for the LDREX_SPEC event.

Arm recommends that this event is implemented if it is not possible to implement the exclusive operation speculatively executed, Store-Exclusive pass, and exclusive operation speculatively executed, Store-Exclusive fail, events with the same degree of speculation as the LDREX_SPEC event.

`0x0070`, **LD_SPEC, Operation speculatively executed, load**

The counter counts each operation counted by LDST_SPEC that is a load operation.

Operations due to Memory-reading instructions are counted as load operations.

It is IMPLEMENTATION DEFINED whether operations due to the prefetch instructions counted by PRF_SPEC are counted as load operations:

- If operations due to the prefetch instructions are counted as load operations, then they are counted by LD_SPEC.
- Otherwise, if operations due to the prefetch instructions are not counted as load operations by LD_SPEC, then it is further IMPLEMENTATION DEFINED which one of the following applies:
  - They are counted as data-processing operations.
  - They are counted by LDST_SPEC.

Arm recommends that if a prefetch operation is not implemented as a NOP and the PRF_SPEC event is not implemented, then the operation is counted as a load operation.

`0x0071`, **ST_SPEC, Operation speculatively executed, store**

The counter counts each operation counted by LDST_SPEC that is a store operation.

Operations due to Memory-writing instructions are counted as store operations.

`0x0072`, **LDST_SPEC, Operation speculatively executed, load or store**

The counter counts each operation counted by INST_SPEC that is a load operation or a store operation.

See LD_SPEC and ST_SPEC for these classifications.

If the PRF_SPEC event is implemented and operations due to the prefetch instructions counted by PRF_SPEC are not counted as load operations by LD_SPEC, then it is IMPLEMENTATION DEFINED whether these operations are counted by LDST_SPEC. Arm recommends that these operations are counted by LDST_SPEC.

`0x0073`, **DP_SPEC, Operation speculatively executed, integer data processing**

The counter counts each operation counted by INST_SPEC that is an integer data-processing operation.

An integer data-processing operation is any operation that is not counted as any of the following:

- A load or store operation, counted by LDST_SPEC.
- A Software change of PC operation, counted by PC_WRITE_SPEC.
- A scalar floating-point data processing operation, counted by VFP_SPEC.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- An Advanced SIMD or SVE data processing operation, counted by ASE_SVE_SPEC.
- A cryptographic operation, counted by CRYPTO_SPEC.

That is, operations due to the following instructions are counted as integer data-processing operations:

- In AArch64 state instructions from the following sections:
  - Data processing - immediate.
  - Data processing - register.
  - System register instructions.
  - 'Instructions with register argument'.
  - System instructions other than Memory-writing instructions.
  - Hint instructions.

`0x0074`**, ASE_SPEC, Operation speculatively executed, Advanced SIMD data processing**

The counter counts each operation counted by INST_SPEC that is an Advanced SIMD data-processing operation.

Operations due to the following instructions are counted as Advanced SIMD data-processing operations:

- For AArch64 state:
  - The SIMD operations listed in Data processing - SIMD and floating-point.
  - Advanced SIMD scalar instructions, including those which operate on floating-point values.
  - If FEAT_Crypto is implemented, the Advanced SIMD `PMULL` and `PMULL2` (1Q variants) instructions. See The Cryptographic Extension in AArch64 state.

This includes all operations that operate on the SIMD&FP registers, except those that are counted as one of the following:

- Integer data-processing operations.
- Scalar floating-point data-processing operations.
- Load or store operations.
- Cryptographic data-processing operations counted by CRYPTO_SPEC. CRYPTO_SPEC does not count the Cryptographic data-processing operations included above.

`0x0075`**, VFP_SPEC, Operation speculatively executed, scalar floating-point data processing**

The counter counts each operation counted by INST_SPEC that is a scalar floating-point operation.

Operations due to the following instructions are counted as scalar floating-point data-processing operations:

- In AArch64 state:
  - The scalar floating-point operations from Data processing - SIMD and floating-point. Only the scalar operations are counted. SIMD floating-point operations are not counted.
  - Operations that take both an integer register and a floating-point register argument and perform a type conversion (to/from integer or to/from fixed-point): `FCVT{<mode>}`, `UCVTF`, and `SCVTF`.

This includes all scalar operations that operate on the SIMD&FP registers as floating-point values, except those that are counted as one of the following:

- SIMD scalar operations.
- Integer data-processing operations.
- Load or store operations.

`0x0076`**, PC_WRITE_SPEC, Operation speculatively executed, Software change of the PC**

The counter counts each operation counted by INST_SPEC that is a Software change of the PC.

Software change of the PC operations are defined by the PC_WRITE_RETIRED event.

It is IMPLEMENTATION DEFINED whether the counter counts the operation when the branch is not taken.

`0x0077`**, CRYPTO_SPEC, Operation speculatively executed, cryptographic data processing**

The counter counts each operation counted by INST_SPEC that is a cryptographic operation.

Operations due to the following instructions are counted as cryptographic data-processing operations:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- In AArch64 state, instructions from The Cryptographic Extension in AArch64 state other than the include Advanced SIMD `PMULL` and `PMULL2` (1Q variants) instructions.

`0x0078`, **BR_IMMED_SPEC, Branch speculatively executed, immediate branch**

The counter counts each speculatively executed immediate branch instruction.

Operations due to the following instructions are counted as speculatively executed immediate branch instructions:

- For AArch64 state, the following instructions:
  - `B <label>`.
  - `B.cond <label>`.
  - `BL <label>`.
  - `CBZ <Rn>, <label>`.
  - `CBNZ <Rn>, <label>`.
  - `TBZ <Rn>, <label>`.
  - `TBNZ <Rn>, <label>`.

Note

Conditional branches are always counted, regardless of whether the branch is taken or not taken.

If the Context synchronization barrier instruction ISB is counted as a Software change of the PC instruction by PC_WRITE_SPEC, then it is IMPLEMENTATION DEFINED whether ISB is counted as a speculatively executed immediate branch instruction.

`0x0079`, **BR_RETURN_SPEC, Branch speculatively executed, procedure return**

The counter counts each speculatively executed procedure return instruction.

Operations due to the following instructions are counted as speculatively executed procedure return instructions:

- For AArch64 state, the following instructions:
  - `RET`.
  - If FEAT_PAuth is implemented, `RETAA` and `RETAB`.

`0x007A`, **BR_INDIRECT_SPEC, Branch speculatively executed, indirect branch**

The counter counts each speculatively executed indirect branch instruction.

Indirect branch instructions are Software change of the PC instructions other than exception-generating instructions and immediate branch instructions. Software change of the PC has the same definition as for the PC_WRITE_SPEC event. Immediate branch instructions are defined by the BR_IMMED_SPEC event.

`0x007C`, **ISB_SPEC, Barrier speculatively executed, ISB**

The counter counts each speculatively executed Instruction Synchronization Barrier instruction.

`0x007D`, **DSB_SPEC, Barrier speculatively executed, DSB**

The counter counts each speculatively executed data synchronization and speculative load barrier instruction.

`0x007E`, **DMB_SPEC, Barrier speculatively executed, DMB**

The counter counts each speculatively executed data memory barrier instruction.

This does not include the implied barrier operations of load/store operations with release consistency semantics.

`0x007F`, **CSDB_SPEC, Barrier speculatively executed, CSDB**

The counter counts each speculatively executed control speculation barrier instruction.

`0x0090`, **RC_LD_SPEC, Release consistency operation speculatively executed, Load-Acquire**

The counter counts each speculatively executed Memory-read operation with acquire or acquirepc semantics.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x0091`, **RC_ST_SPEC, Release consistency operation speculatively executed, Store-Release**

The counter counts each speculatively executed Memory-write operation with release semantics.

`0x00A0`, **L3D_CACHE_RD, Level 3 data cache access, read**

If the L3D_CACHE_RW event is implemented, the counter counts each access counted by L3D_CACHE_RW that is a Memory-read operation.

If the L3D_CACHE_RW event is not implemented, the counter counts each access counted by L3D_CACHE that is a Memory-read operation.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x00A1`, **L3D_CACHE_WR, Level 3 data cache access, write**

If the L3D_CACHE_RW event is implemented, the counter counts each access counted by L3D_CACHE_RW that is a Memory-write operation.

If the L3D_CACHE_RW event is not implemented, the counter counts each access counted by L3D_CACHE that is a Memory-write operation.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x00A2`, **L3D_CACHE_REFILL_RD, Level 3 data cache refill, read**

The counter counts each access counted by both L3D_CACHE_RD and L3D_CACHE_REFILL.

That is, every refill of the Level 3 data or unified cache counted by L3D_CACHE_REFILL that is caused by a Memory-read operation.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x00A3`, **L3D_CACHE_REFILL_WR, Level 3 data cache refill, write**

The counter counts each access counted by both L3D_CACHE_REFILL and L3D_CACHE_WR.

That is, every refill of the Level 3 data or unified cache counted by L3D_CACHE_REFILL that is caused by a Memory-write operation.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x00A6`, **L3D_CACHE_WB_VICTIM, Level 3 data cache write-back, victim**

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The counter counts each write-back counted by L3D_CACHE_WB that occurs because the line is allocated for an access made by the PE.

`0x00A7`, **L3D_CACHE_WB_CLEAN, Level 3 data cache write-back, cleaning and coherency**

The counter counts each write-back counted by L3D_CACHE_WB that occurs because of a coherency operation made by another PE or, optionally, the execution of a cache maintenance instruction.

Whether write-backs that are caused by the execution of a cache maintenance instruction are counted is IMPLEMENTATION DEFINED.

Note

The transfer of a dirty cache line from the Level 3 data cache of this PE to the data cache of another PE due to a hardware coherency operation is not counted unless the dirty cache line is also written back to a lower level cache or memory.

If a coherency request from a requestor outside of the PE results in a write-back, it is an Unattributable event.

`0x00A8`, **L3D_CACHE_INVAL, Level 3 data cache invalidate**

The counter counts each invalidation of a cache line in the Level 3 data or unified cache. For example:

- Invalidation of a cache line because of a cache maintenance operation.
- Transfer of ownership of a cache line to another cache because of a coherency or refill request.

The counter does not count events if a cache refill of the Level 3 data or unified cache invalidates a line in the Level 3 data or unified cache.

If FEAT_PMUv3p4 is not implemented, the counter does not count locally-executed cache maintenance instructions that operate by set/way.

If FEAT_PMUv3p4 is implemented, it is IMPLEMENTATION DEFINED whether the counter counts locally-executed cache maintenance instructions that operate by set/way.

Note

Software that uses this event must know whether the Level 3 data cache is shared with other PEs. This event does not follow the general rule of Level 3 data cache events of only counting Attributable events.

`0x4000`, **SAMPLE_POP, Statistical Profiling sample population**

The counter counts each operation that might be sampled, whether or not the operation was sampled.

The counter does not count when profiling is disabled.

PMCEID0_EL0[32] reads as 1 if this event is implemented and 0 otherwise.

`0x4001`, **SAMPLE_FEED, Statistical Profiling sample taken**

The counter counts each time the sample interval counter reaches zero and is reloaded, and the sample does not collide with the previous sample.

Samples that are removed by filtering, or discarded, and not written to the Profiling Buffer are counted.

PMCEID0_EL0[33] reads as 1 if this event is implemented and 0 otherwise.

`0x4002`, **SAMPLE_FILTRATE, Statistical Profiling sample taken and not removed by filtering**

The counter counts each sample counted by SAMPLE_FEED that is not removed by filtering.

Sample records that are not removed by filtering, but are discarded before being written to the Profiling Buffer because of a Profiling Buffer management event, are counted.

PMCEID0_EL0[34] reads as 1 if this event is implemented and 0 otherwise.

`0x4003`, **SAMPLE_COLLISION, Statistical Profiling sample collided with previous sample**

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The counter counts each time the sample interval counter reaches zero and is reloaded, and the sample collides with a previous sample.

For more information, see ‘Sample collisions’.

PMCEID0_EL0[35] reads as 1 if this event is implemented and 0 otherwise.

`0x4004`**, CNT_CYCLES, Constant frequency cycles**

The counter increments at a constant frequency equal to the rate of increment of the System Counter, CNTPCT_EL0.

PMCEID0_EL0[36] reads as 1 if this event is implemented and 0 otherwise.

`0x4005`**, STALL_BACKEND_MEM, Memory stall cycles**

The counter counts each cycle counted by STALL_BACKEND_MEMBOUND where there is a demand data miss in the last level of data or unified cache within the PE clock domain.

It is IMPLEMENTATION DEFINED whether the counter counts backend stall cycles when a non-cacheable data access is in progress.

PMCEID0_EL0[37] reads as 1 if this event is implemented and 0 otherwise.

`0x4006`**, L1I_CACHE_LMISS, Level 1 instruction cache long-latency miss**

If the L1I_CACHE_RD event is implemented, the counter counts each access to the Level 1 instruction or unified cache counted by L1I_CACHE_RD that incurs additional latency because it returns instructions from outside of the Level 1 instruction or unified cache of this PE.

If the L1I_CACHE_RD event is not implemented, the counter counts each access to the Level 1 instruction or unified cache counted by L1I_CACHE that incurs additional latency because it returns instructions from outside of the Level 1 instruction or unified cache of this PE.

The event indicates to software that the access missed in the Level 1 instruction or unified cache and might have a significant performance impact due to the additional latency compared to the latency of an access that hits in the Level 1 instruction or unified cache.

The counter does not count:

- Accesses where the additional latency is unlikely to be significantly performance-impacting. For example, if the access hits in another cache in the same local cluster, and the additional latency is small when compared to a miss in all Level 1 caches that the access looks up in and results in instructions being returned from a Level 2 cache or elsewhere beyond the Level 1 instruction or unified cache.
- A miss that does not cause a new cache refill but is satisfied from a previous miss.

An implementation is not required to measure the latency, nor to track the access to determine whether the additional latency caused a performance impact. An implementation can extend the definition of this event with additional scenarios where an access might have a significant performance impact due to additional latency for the access.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance operations are counted.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID0_EL0[38] reads as 1 if this event is implemented and 0 otherwise.

`0x4009`**, L2D_CACHE_LMISS_RD, Level 2 data cache long-latency read miss**

The counter counts each Memory-read operation to the Level 2 data or unified cache counted by L2D_CACHE that incurs additional latency because it returns data from outside of the Level 1 and Level 2 data or unified caches of this PE.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The event indicates to software that the access missed in the Level 2 data or unified cache and might have a significant performance impact due to the additional latency compared to the latency of an access that hits in the Level 2 data or unified cache.

The counter does not count:

- Accesses where the additional latency is unlikely to be significantly performance-impacting. For example, if the access hits in another cache in the same local cluster, and the additional latency is small when compared to a miss in all Level 2 caches that the access looks up in and results in an access being made to a Level 3 cache or elsewhere beyond the Level 2 data or unified cache. This might be counted as a Level 1 cache miss.
- A miss that does not cause a new cache refill but is satisfied from a previous miss.

An implementation is not required to measure the latency, nor to track the access to determine whether the additional latency caused a performance impact. An implementation can extend the definition of this event with additional scenarios where an access might have a significant performance impact due to additional latency for the access.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance operations are counted.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID0_EL0[41] reads as 1 if this event is implemented and 0 otherwise.

`0x400A`**, L2I_CACHE_LMISS, Level 2 instruction cache long-latency miss**

If the L2I_CACHE_RD event is implemented, the counter counts each access to the Level 2 instruction or unified cache counted by L2I_CACHE_RD that incurs additional latency because it returns instructions from outside of the Level 1 and Level 2 instruction or unified caches of this PE.

If the L2I_CACHE_RD event is not implemented, the counter counts each access to the Level 2 instruction or unified cache counted by L2I_CACHE that incurs additional latency because it returns instructions from outside of the Level 1 and Level 2 instruction or unified caches of this PE.

The event indicates to software that the access missed in the Level 2 instruction or unified cache and might have a significant performance impact due to the additional latency compared to the latency of an access that hits in the Level 2 instruction or unified cache.

The counter does not count:

- Accesses where the additional latency is unlikely to be significantly performance-impacting. For example, if the access hits in another cache in the same local cluster, and the additional latency is small when compared to a miss in all Level 2 caches that the access looks up in and results in instructions being returned from a Level 3 cache or elsewhere beyond the Level 2 instruction or unified cache. This might be counted as a Level 1 cache miss.
- A miss that does not cause a new cache refill but is satisfied from a previous miss.

An implementation is not required to measure the latency, nor to track the access to determine whether the additional latency caused a performance impact. An implementation can extend the definition of this event with additional scenarios where an access might have a significant performance impact due to additional latency for the access.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance operations are counted.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID0_EL0[42] reads as 1 if this event is implemented and 0 otherwise.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x400B`**, L3D_CACHE_LMISS_RD, Level 3 data cache long-latency read miss**

The counter counts each Memory-read operation to the Level 3 data or unified cache counted by L3D_CACHE that incurs additional latency because it returns data from outside of the Level 1 to Level 3 data or unified caches of this PE.

The event indicates to software that the access missed in the Level 3 data or unified cache and might have a significant performance impact due to the additional latency compared to the latency of an access that hits in the Level 3 data or unified cache.

The counter does not count:

- Accesses where the additional latency is unlikely to be significantly performance-impacting. For example, if the access hits in another cache in the same local cluster, and the additional latency is small when compared to a miss in all Level 3 caches that the access looks up in and results in an access being made to a lower level cache or elsewhere beyond the Level 3 data or unified cache. This might be counted as a Level 2 cache miss.
- A miss that does not cause a new cache refill but is satisfied from a previous miss.

An implementation is not required to measure the latency, nor to track the access to determine whether the additional latency caused a performance impact. An implementation can extend the definition of this event with additional scenarios where an access might have a significant performance impact due to additional latency for the access.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance operations are counted.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

PMCEID0_EL0[43] reads as 1 if this event is implemented and 0 otherwise.

`0x4010`**, TRCEXTOUT0, Trace unit external output 0**

The counter counts each event signaled by the trace unit on external event 0.

It is IMPLEMENTATION DEFINED whether this event is available as an external input to the ETE.

PMCEID0_EL0[48] reads as 1 if this event is implemented and 0 otherwise.

`0x4011`**, TRCEXTOUT1, Trace unit external output 1**

The counter counts each event signaled by the trace unit on external event 1.

It is IMPLEMENTATION DEFINED whether this event is available as an external input to the ETE.

PMCEID0_EL0[49] reads as 1 if this event is implemented and 0 otherwise.

`0x4012`**, TRCEXTOUT2, Trace unit external output 2**

The counter counts each event signaled by the trace unit on external event 2.

It is IMPLEMENTATION DEFINED whether this event is available as an external input to the ETE.

PMCEID0_EL0[50] reads as 1 if this event is implemented and 0 otherwise.

`0x4013`**, TRCEXTOUT3, Trace unit external output 3**

The counter counts each event signaled by the trace unit on external event 3.

It is IMPLEMENTATION DEFINED whether this event is available as an external input to the ETE.

PMCEID0_EL0[51] reads as 1 if this event is implemented and 0 otherwise.

`0x4018`**, CTI_TRIGOUT4, Cross-trigger Interface output trigger 4**

The counter counts each event signaled on CTI output trigger 4.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

CTI output triggers are input events to the PMU and trace unit.

PMCEID0_EL0[56] reads as 1 if this event is implemented and 0 otherwise.

`0x4019`, **CTI_TRIGOUT5, Cross-trigger Interface output trigger 5**

The counter counts each event signaled on CTI output trigger 5.

Note

CTI output triggers are input events to the PMU and trace unit.

PMCEID0_EL0[57] reads as 1 if this event is implemented and 0 otherwise.

`0x401A`, **CTI_TRIGOUT6, Cross-trigger Interface output trigger 6**

The counter counts each event signaled on CTI output trigger 6.

Note

CTI output triggers are input events to the PMU and trace unit.

PMCEID0_EL0[58] reads as 1 if this event is implemented and 0 otherwise.

`0x401B`, **CTI_TRIGOUT7, Cross-trigger Interface output trigger 7**

The counter counts each event signaled on CTI output trigger 7.

Note

CTI output triggers are input events to the PMU and trace unit.

PMCEID0_EL0[59] reads as 1 if this event is implemented and 0 otherwise.

`0x4020`, **LDST_ALIGN_LAT, Access with additional latency from alignment**

The counter counts each access counted by MEM_ACCESS that incurred additional latency due to the alignment of the address and size of data being accessed.

PMCEID1_EL0[32] reads as 1 if this event is implemented and 0 otherwise.

`0x4021`, **LD_ALIGN_LAT, Load with additional latency from alignment**

The counter counts each Memory-read operation counted by LDST_ALIGN_LAT.

PMCEID1_EL0[33] reads as 1 if this event is implemented and 0 otherwise.

`0x4022`, **ST_ALIGN_LAT, Store with additional latency from alignment**

The counter counts each Memory-write operation counted by LDST_ALIGN_LAT.

PMCEID1_EL0[34] reads as 1 if this event is implemented and 0 otherwise.

`0x4032`, **TME_TRANSACTION_FAILED, Transaction failed or was canceled**

The counter counts each transaction that fails or is canceled.

If PMEVTYPER<n>_EL0.T is 1, it is IMPLEMENTATION DEFINED whether or not the transaction is counted by the counter.

PMCEID1_EL0[50] reads as 1 if this event is implemented and 0 otherwise.

`0x4034`, **TME_INST_RETIRED_COMMITTED, Instruction architecturally executed, in a committed transaction**

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The counter counts each Instruction architecturally executed counted by INST_RETIRED when the PE is in Transactional state and the currently executing outer transaction commits.

It is IMPLEMENTATION DEFINED whether the counter counts more than 2^{32}-1 instructions in a single outer transaction.

Note

Two example implementations of this event are as follows:

- The implementation accumulates events to the counter directly. If the transaction fails, the counter is restored to the value it had when the transaction started.
- The implementation accumulates events without updating the counter. If the transaction commits, the counter is updated with the accumulated value.

The counter does not count the instruction if the transaction fails or is canceled.

PMCEID1_EL0[52] reads as 1 if this event is implemented and 0 otherwise.

`0x4035`**, TME_CPU_CYCLES_COMMITTED, PE cycle, in a committed transaction**

The counter counts each PE cycle counted by CPU_CYCLES when the PE is in Transactional state and the currently executing outer transaction commits.

It is IMPLEMENTATION DEFINED whether the counter counts more than 2^{32}-1 cycles in a single outer transaction.

Note

Two example implementations of this event are as follows:

- The implementation accumulates events to the counter directly. If the transaction fails, the counter is restored to the value it had when the transaction started.
- The implementation accumulates events without updating the counter. If the transaction commits, the counter is updated with the accumulated value.

The counter does not count the cycle if the transaction fails or is canceled.

PMCEID1_EL0[53] reads as 1 if this event is implemented and 0 otherwise.

`0x4038`**, TME_FAILURE_CNCL, Transaction failed with CNCL cause**

The counter counts each transaction counted by TME_TRANSACTION_FAILED that fails with CNCL cause.

If PMEVTYPER<n>_EL0.T is 1, it is IMPLEMENTATION DEFINED whether or not the transaction is counted by the counter.

PMCEID1_EL0[56] reads as 1 if this event is implemented and 0 otherwise.

`0x4039`**, TME_FAILURE_NEST, Transaction failed with NEST cause**

The counter counts each transaction counted by TME_TRANSACTION_FAILED that fails with NEST cause.

If PMEVTYPER<n>_EL0.T is 1, it is IMPLEMENTATION DEFINED whether or not the transaction is counted by the counter.

PMCEID1_EL0[57] reads as 1 if this event is implemented and 0 otherwise.

`0x403A`**, TME_FAILURE_ERR, Transaction failed with ERR cause**

The counter counts each transaction counted by TME_TRANSACTION_FAILED that fails with ERR cause.

If PMEVTYPER<n>_EL0.T is 1, it is IMPLEMENTATION DEFINED whether or not the transaction is counted by the counter.

PMCEID1_EL0[58] reads as 1 if this event is implemented and 0 otherwise.

`0x403B`**, TME_FAILURE_IMP, Transaction failed with IMP cause**

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The counter counts each transaction counted by TME_TRANSACTION_FAILED that fails with IMP cause.

If PMEVTYPER<n>_EL0.T is 1, it is IMPLEMENTATION DEFINED whether or not the transaction is counted by the counter.

PMCEID1_EL0[59] reads as 1 if this event is implemented and 0 otherwise.

`0x403C`, **TME_FAILURE_MEM, Transaction failed with MEM cause**

The counter counts each transaction counted by TME_TRANSACTION_FAILED that fails with MEM cause.

If PMEVTYPER<n>_EL0.T is 1, it is IMPLEMENTATION DEFINED whether or not the transaction is counted by the counter.

PMCEID1_EL0[60] reads as 1 if this event is implemented and 0 otherwise.

`0x403D`, **TME_FAILURE_SIZE, Transaction failed with SIZE cause**

The counter counts each transaction counted by TME_TRANSACTION_FAILED that fails with SIZE cause.

If PMEVTYPER<n>_EL0.T is 1, it is IMPLEMENTATION DEFINED whether or not the transaction is counted by the counter.

PMCEID1_EL0[61] reads as 1 if this event is implemented and 0 otherwise.

`0x403E`, **TME_FAILURE_TLBI, Transaction failed due to execution of TLBI by another PE**

The counter counts each transaction counted by TME_TRANSACTION_FAILED that fails with IMP cause due to the execution of a `TLBI` instruction by another PE.

If PMEVTYPER<n>_EL0.T is 1, it is IMPLEMENTATION DEFINED whether or not the transaction is counted by the counter.

PMCEID1_EL0[62] reads as 1 if this event is implemented and 0 otherwise.

`0x403F`, **TME_FAILURE_WSET, Transaction failed due to transactional write set limit overflow**

The counter counts each transaction counted by TME_TRANSACTION_FAILED that fails with SIZE cause due to a memory access that causes an eviction of an entry from the transactional write set.

If PMEVTYPER<n>_EL0.T is 1, it is IMPLEMENTATION DEFINED whether or not the transaction is counted by the counter.

PMCEID1_EL0[63] reads as 1 if this event is implemented and 0 otherwise.

`0x8004`, **SIMD_INST_SPEC, Operation speculatively executed, SIMD**

The counter counts each speculatively executed operation due to any of:

- An Advanced SIMD operation. Non-SIMD data-processing operations are not counted.
- A structure load/store of one or more SIMD&FP registers.
- A load and replicate to one or more SIMD&FP registers.
- A scalar load/store of a SIMD&FP Q register or pair of Q registers.
- An SVE load/store.

It is IMPLEMENTATION DEFINED whether scalar loads and stores to SIMD&FP registers other that those listed above are counted.

`0x8005`, **ASE_INST_SPEC, Operation speculatively executed, Advanced SIMD**

The counter counts each speculatively executed operation counted by ASE_SVE_INST_SPEC that is classified as an Advanced SIMD operation.

Operations due to the following instructions are counted as Advanced SIMD operations:

- For AArch64 state:
  - The SIMD operations listed in Data processing - SIMD and floating-point.
  - If FEAT_Crypto is implemented, the Advanced SIMD Cryptographic instructions. See The Cryptographic Extension in AArch64 state.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Advanced SIMD loads and stores.

This includes all operations that operate on the SIMD&FP registers, except those that are counted as one of the following:

- Integer data-processing operations.
- Scalar floating-point data-processing operations.

It is IMPLEMENTATION DEFINED whether Advanced SIMD scalar operations are counted as Advanced SIMD operations. This includes Advanced SIMD scalar instructions which operate on floating-point values.

`0x8007`, **ASE_SVE_INST_SPEC, Operation speculatively executed, Advanced SIMD or SVE**

The counter counts each speculatively executed operation counted by SE_INST_SPEC that is classified as an Advanced SIMD operation or an SVE operation.

See ASE_INST_SPEC and SVE_INST_SPEC for these classifications.

`0x8008`, **UOP_SPEC, Microarchitectural operation speculatively executed**

The counter counts each speculatively executed microarchitectural operation.

`0x8009`, **ASE_UOP_SPEC, Microarchitectural operation speculatively executed, Advanced SIMD**

The counter counts each speculatively executed microarchitectural operation counted by ASE_SVE_UOP_SPEC due to an A64 Advanced SIMD instruction.

It is IMPLEMENTATION DEFINED whether the counter counts microarchitectural operations due to Advanced SIMD scalar instructions.

`0x800B`, **ASE_SVE_UOP_SPEC, Microarchitectural operation speculatively executed, Advanced SIMD or SVE**

The counter counts each speculatively executed microarchitectural operation counted by UOP_SPEC due to an A64 Advanced SIMD instruction.

It is IMPLEMENTATION DEFINED whether the counter counts microarchitecural operations due to Advanced SIMD scalar instructions.

`0x800C`, **SIMD_UOP_SPEC, Microarchitectural operation speculatively executed, SIMD**

The counter counts each speculatively executed microarchitectural operation counted by UOP_SPEC due to any of:

- An SVE instruction other than non-SIMD SVE instructions.
- An A64 Advanced SIMD instruction other than an Advanced SIMD scalar instruction.

`0x8010`, **FP_SPEC, Floating-point operation speculatively executed, including SIMD**

The counter counts each speculatively executed floating-point data-processing operation due to any of:

- An A64 scalar instruction.
- An A64 Advanced SIMD instruction.

It is IMPLEMENTATION DEFINED whether the counter counts any operation due to the following instructions:

- Scalar: `FABS`, or `FNEG (scalar)`.
- Advanced SIMD: `FABS`, `FNEG (vector)`, `FRECPE`, or `FRSQRTE`.
- Data movement instructions which move data without carrying out any operations on the data. This includes identity data-processing operations.
- Conditional select instructions.
- Floating-point conversion instructions.

Any operation counted by INT_SPEC is not counted.

Arm recommends that if FP_CVT_SPEC is implemented, FP_SPEC counts operations due to convert instructions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

This event differs from the VFP_SPEC event which does not count SIMD operations.

`0x8011`, **ASE_FP_SPEC, Floating-point operation speculatively executed, Advanced SIMD**

The counter counts each speculatively executed floating-point operation counted by ASE_SVE_FP_SPEC due to an A64 Advanced SIMD instruction.

`0x8013`, **ASE_SVE_FP_SPEC, Floating-point operation speculatively executed, Advanced SIMD or SVE**

The counter counts each speculatively executed floating-point operation counted by SE_FP_SPEC due to an A64 Advanced SIMD instruction.

`0x8014`, **FP_HP_SPEC, Floating-point operation speculatively executed, half-precision**

The counter counts each speculatively executed floating-point operation counted by FP_SPEC where the largest type is half-precision.

`0x8015`, **ASE_FP_HP_SPEC, Floating-point operation speculatively executed, Advanced SIMD half-precision, data-processing**

The counter counts each speculatively executed half-precision floating-point data-processing operation counted by ASE_SVE_FP_HP_SPEC due to an Advanced SIMD instruction.

`0x8017`, **ASE_SVE_FP_HP_SPEC, Floating-point operation speculatively executed, Advanced SIMD or SVE half-precision**

The counter counts each speculatively executed half-precision floating-point operation counted by FP_HP_SPEC due to an Advanced SIMD.

`0x8018`, **FP_SP_SPEC, Floating-point operation speculatively executed, single-precision**

The counter counts each speculatively executed floating-point operation counted by FP_SPEC where the largest type is single-precision.

`0x8019`, **ASE_FP_SP_SPEC, Floating-point operation speculatively executed, Advanced SIMD single-precision, data-processing**

The counter counts each speculatively executed single-precision floating-point data-processing operation counted by ASE_SVE_FP_SP_SPEC due to an Advanced SIMD instruction.

`0x801B`, **ASE_SVE_FP_SP_SPEC, Floating-point operation speculatively executed, Advanced SIMD or SVE single-precision, data-processing**

The counter counts each speculatively executed single-precision floating-point data-processing operation counted by FP_SP_SPEC due to an Advanced SIMD.

`0x801C`, **FP_DP_SPEC, Floating-point operation speculatively executed, double-precision, data-processing**

The counter counts each speculatively executed floating-point data-processing operation counted by FP_SPEC where the largest type is double-precision.

`0x801D`, **ASE_FP_DP_SPEC, Floating-point operation speculatively executed, Advanced SIMD double-precision, data-processing**

The counter counts each speculatively executed double-precision floating-point data-processing operation counted by ASE_SVE_FP_DP_SPEC due to an Advanced SIMD instruction.

`0x801F`, **ASE_SVE_FP_DP_SPEC, Floating-point operation speculatively executed, Advanced SIMD or SVE double-precision, data-processing**

The counter counts each speculatively executed double-precision floating-point data-processing operation due to an Advanced SIMD.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x8020`, **FP_DIV_SPEC, Floating-point operation speculatively executed, divide**

The counter counts each speculatively executed floating-point divide operation counted by FP_SPEC due to any of the following A64 instructions:

- Scalar: FDIV.
- Advanced SIMD: FDIV.

`0x8021`, **ASE_FP_DIV_SPEC, Floating-point operation speculatively executed, Advanced SIMD divide**

The counter counts each speculatively executed floating-point divide operation counted by ASE_SVE_FP_DIV_SPEC due to any of the following A64 instructions:

- Advanced SIMD: FDIV.

`0x8023`, **ASE_SVE_FP_DIV_SPEC, Floating-point operation speculatively executed, Advanced SIMD or SVE divide**

The counter counts each speculatively executed floating-point divide operation counted by FP_DIV_SPEC due to any of the following A64 instructions:

- Advanced SIMD: FDIV.

`0x8024`, **FP_SQRT_SPEC, Floating-point operation speculatively executed, square root**

The counter counts each speculatively executed floating-point square-root operation counted by FP_SPEC due to any of the following A64 instructions:

- Scalar: FSQRT.
- Advanced SIMD: FSQRT.

`0x8025`, **ASE_FP_SQRT_SPEC, Floating-point operation speculatively executed, Advanced SIMD square root**

The counter counts each speculatively executed floating-point square-root operation counted by ASE_SVE_FP_SQRT_SPEC due to any of the following A64 instructions:

- Advanced SIMD: FSQRT.

`0x8027`, **ASE_SVE_FP_SQRT_SPEC, Floating-point operation speculatively executed, Advanced SIMD or SVE square-root**

The counter counts each speculatively executed floating-point square-root operation counted by FP_SQRT_SPEC due to any of the following A64 instructions:

- Advanced SIMD: FSQRT.

`0x8028`, **FP_FMA_SPEC, Floating-point operation speculatively executed, FMA**

The counter counts each speculatively executed floating-point fused multiply-add or multiply-subtract operation counted by FP_SPEC due to any of the following A64 instructions:

- Scalar: FMADD, FMSUB, FNMADD, or FNMSUB.
- Advanced SIMD: FMLA or FMLS.
- When FEAT_FCMA is implemented, Advanced SIMD: FCMLA.
- When FEAT_FHM is implemented, Advanced SIMD: FMLAL, FMLAL2, FMLSL, or FMLSL2.

`0x8029`, **ASE_FP_FMA_SPEC, Floating-point operation speculatively executed, Advanced SIMD FMA**

The counter counts each speculatively executed floating-point fused multiply-add or multiply-subtract operation counted by ASE_SVE_FP_FMA_SPEC due to any of the following A64 instructions:

- Advanced SIMD: FMLA or FMLS.
- When FEAT_FCMA is implemented, Advanced SIMD: FCMLA.
- When FEAT_FHM is implemented, Advanced SIMD: FMLAL, FMLAL2, FMLSL, or FMLSL2.

`0x802B`, **ASE_SVE_FP_FMA_SPEC, Floating-point operation speculatively executed, Advanced SIMD or SVE FMA**

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The counter counts each speculatively executed floating-point fused multiply-add or multiply-subtract operation counted by FP_FMA_SPEC due to any of the following A64 instructions:

- Advanced SIMD: FMLA or FMLS.
- When FEAT_FCMA is implemented, Advanced SIMD: FCMLA.
- When FEAT_FHM is implemented, Advanced SIMD: FMLAL, FMLAL2, FMLSL, or FMLSL2.

`0x802C`, **FP_MUL_SPEC, Floating-point operation speculatively executed, multiply**

The counter counts each speculatively executed floating-point multiply operation counted by FP_SPEC due to any of the following A64 instructions:

- Scalar: FMUL or FMULX.
- Advanced SIMD: FMUL or FMULX.

`0x802D`, **ASE_FP_MUL_SPEC, Floating-point operation speculatively executed, Advanced SIMD multiply**

The counter counts each speculatively executed floating-point multiply operation counted by ASE_SVE_FP_MUL_SPEC due to any of the following A64 instructions:

- Advanced SIMD: FMUL or FMULX.

`0x802F`, **ASE_SVE_FP_MUL_SPEC, Floating-point operation speculatively executed, Advanced SIMD or SVE multiply**

The counter counts each speculatively executed floating-point fused multiply-add or multiply-subtract operation counted by FP_MUL_SPEC due to any of the following A64 instructions:

- Advanced SIMD: FMUL or FMULX.

`0x8030`, **FP_ADDSUB_SPEC, Floating-point operation speculatively executed, add or subtract**

The counter counts each speculatively executed floating-point add or subtract operation counted by FP_SPEC due to any of the following A64 instructions:

- Scalar: FADD or FSUB.
- Advanced SIMD: FABD, FADD, or FSUB.

`0x8031`, **ASE_FP_ADDSUB_SPEC, Floating-point operation speculatively executed, Advanced SIMD add or subtract**

The counter counts each speculatively executed floating-point add or subtract operation counted by ASE_SVE_FP_ADDSUB_SPEC due to any of the following A64 instructions:

- Advanced SIMD: FABD, FADD, or FSUB.

`0x8033`, **ASE_SVE_FP_ADDSUB_SPEC, Floating-point operation speculatively executed, Advanced SIMD or SVE add or subtract**

The counter counts each speculatively executed floating-point add or subtract operation counted by FP_ADDSUB_SPEC due to any of the following A64 instructions:

- Advanced SIMD: FABD, FADD, or FSUB.

`0x8034`, **FP_RECPE_SPEC, Floating-point operation speculatively executed, reciprocal estimate**

The counter counts each speculatively executed floating-point reciprocal estimate operation counted by FP_SPEC due to any of the following A64 instructions:

- Advanced SIMD: FRECPE or FRSQRTE.

`0x8035`, **ASE_FP_RECPE_SPEC, Floating-point operation speculatively executed, Advanced SIMD reciprocal estimate**

The counter counts each speculatively executed floating-point reciprocal estimate operation counted by ASE_SVE_FP_RECPE_SPEC due to any of the following A64 instructions:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Advanced SIMD: FRECPE or FRSQRTE.

`0x8037`, **ASE_SVE_FP_RECPE_SPEC, Floating-point operation speculatively executed, Advanced SIMD or SVE reciprocal estimate**

The counter counts each speculatively executed floating-point reciprocal estimate operation counted by FP_RECPE_SPEC due to any of the following A64 instructions:

- Advanced SIMD: FRECPE or FRSQRTE.

`0x8038`, **FP_CVT_SPEC, Floating-point operation speculatively executed, convert**

The counter counts each speculatively executed floating-point convert operation due to any of:

- An A64 scalar floating-point conversion instruction.
- An Advanced SIMD floating-point conversion instruction.
- An SVE floating-point conversion instruction.

This includes both conversions between floating-point types, and conversions between integer and floating-point types.

`0x8039`, **ASE_FP_CVT_SPEC, Floating-point operation speculatively executed, Advanced SIMD convert**

The counter counts each speculatively executed floating-point convert operation counted by ASE_SVE_FP_CVT_SPEC due to an A64 Advanced SIMD instruction.

`0x803B`, **ASE_SVE_FP_CVT_SPEC, Floating-point operation speculatively executed, Advanced SIMD or SVE convert**

The counter counts each speculatively executed floating-point convert operation counted by FP_CVT_SPEC due to an A64 Advanced SIMD instruction.

`0x803D`, **ASE_FP_PREDUCE_SPEC, Floating-point operation speculatively executed, Advanced SIMD pairwise add step or pairwise reduce step**

The counter counts each speculatively executed floating-point pairwise add step or pairwise reduce step operation counted by ASE_FP_VREDUCE_SPEC due to any of the following A64 instructions:

- Advanced SIMD: FADDP, FMAXNMP, FMAXP, FMINNMP, or FMINP.

`0x803F`, **ASE_SVE_FP_VREDUCE_SPEC, Floating-point operation speculatively executed, Advanced SIMD or SVE vector reduction**

The counter counts each speculatively executed floating-point reduction operation counted by ASE_SVE_FP_SPEC due to any of the following A64 instructions:

- Advanced SIMD: FADDP, FMAXNMP, FMAXNMV, FMAXP, FMAXV, FMINNMP, FMINNMV, FMINP, or FMINV.

`0x8040`, **INT_SPEC, Integer operation speculatively executed**

The counter counts each speculatively executed integer arithmetic operation due to any of:

- An A64 scalar data-processing instruction.
- An Advanced SIMD instruction.
- An SVE data-processing instruction.

It is IMPLEMENTATION DEFINED whether the counter counts any operation due to the following instructions:

- Scalar: `FABS`, `FNEG (scalar)`.
- Advanced SIMD: `FABS`, `FNEG (vector)`, `FRECPE`, `FRSQRTE`.
- Data movement instructions which move data without carrying out any operations on the data. This includes identity data-processing operations.
- Conditional select instructions.
- Floating-point conversion instructions.

Any operation counted by FP_SPEC is not counted.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x8041`, **ASE_INT_SPEC, Integer operation speculatively executed, Advanced SIMD**

The counter counts each speculatively executed integer operation counted by ASE_SVE_INT_SPEC due to an A64 Advanced SIMD instruction.

`0x8043`, **ASE_SVE_INT_SPEC, Integer operation speculatively executed, Advanced SIMD or SVE**

The counter counts each speculatively executed integer arithmetic operation counted by SE_INT_SPEC due to an A64 Advanced SIMD.

`0x8044`, **INT_DIV_SPEC, Integer operation speculatively executed, divide**

The counter counts each speculatively executed integer divide operation counted by INT_SPEC due to any of the following A64 instructions:

- Scalar: SDIV or UDIV.

`0x8045`, **INT_DIV64_SPEC, Integer operation speculatively executed, 64-bit divide**

The counter counts each speculatively executed integer divide operation counted by INT_SPEC due to any of the following A64 instructions:

- Scalar: SDIV or UDIV.

The counter only counts operations with 64-bit operands or vector elements.

`0x8048`, **INT_MUL_SPEC, Integer operation speculatively executed, multiply**

The counter counts each speculatively executed integer multiply or multiply-accumulate operation counted by INT_SPEC due to any of the following A64 instructions:

- Scalar: MADD, MSUB, MUL, SMADDL, SMULH, UMADDL, or UMULH.
- Advanced SIMD: MLA, MLS, MUL, PMUL, PMULL, PMULL2, SMLAL, SMLAL2, SMLSL, SMLSL2, SMULL, SMULL2, SQDMLAL, SQDMLAL2, SQDMLSL, SQDMLSL2, SQDMULH, SQDMULL, SQDMULL2, SQRDMULH, UMLAL, UMLAL2, UMLSL, UMLSL2, UMULL, or UMULL2.
- When FEAT_RDM is implemented, Advanced SIMD: SQRDMLAH or SQRDMLSH.

`0x8049`, **ASE_INT_MUL_SPEC, Integer operation speculatively executed, Advanced SIMD multiply**

The counter counts each speculatively executed integer multiply or multiply-accumulate operation counted by ASE_SVE_INT_MUL_SPEC due to any of the following A64 instructions:

- Advanced SIMD: MLA, MLS, MUL, PMUL, PMULL, PMULL2, SMLAL, SMLAL2, SMLSL, SMLSL2, SMULL, SMULL2, SQDMLAL, SQDMLAL2, SQDMLSL, SQDMLSL2, SQDMULH, SQDMULL, SQDMULL2, SQRDMULH, UMLAL, UMLAL2, UMLSL, UMLSL2, UMULL, or UMULL2.
- When FEAT_RDM is implemented, Advanced SIMD: SQRDMLAH or SQRDMLSH.

`0x804B`, **ASE_SVE_INT_MUL_SPEC, Integer operation speculatively executed, Advanced SIMD or SVE multiply**

The counter counts each speculatively executed integer multiply or multiply-accumulate operation counted by INT_MUL_SPEC due to any of the following A64 instructions:

- Advanced SIMD: MLA, MLS, MUL, PMUL, PMULL, PMULL2, SMLAL, SMLAL2, SMLSL, SMLSL2, SMULL, SMULL2, SQDMLAL, SQDMLAL2, SQDMLSL, SQDMLSL2, SQDMULH, SQDMULL, SQDMULL2, SQRDMULH, UMLAL, UMLAL2, UMLSL, UMLSL2, UMULL, or UMULL2.
- When FEAT_RDM is implemented, Advanced SIMD: SQRDMLAH or SQRDMLSH.

`0x804C`, **INT_MUL64_SPEC, Integer operation speculatively executed, 64×64 multiply**

The counter counts each speculatively executed 64×64 integer multiply operation counted by INT_SPEC due to any of the following A64 instructions:

- Scalar: MADD, MSUB, MUL, SMULH, or UMULH.

The counter only counts operations that perform a 64-bit × 64-bit integer multiply operations.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x804E`, **INT_MULH64_SPEC, Integer operation speculatively executed, 64×64 multiply returning high part**

The counter counts each speculatively executed widening 64×64 integer multiply operation counted by INT_SPEC due to any of the following A64 instructions:

- Scalar: SMULH or UMULH.

These instructions perform 64-bit × 64-bit integer multiply operations.

`0x8053`, **ASE_SVE_FP_BF16_SPEC, Floating-point operation speculatively executed, SVE or Advanced SIMD BFloat16, data-processing**

The counter counts each speculatively executed BFloat16 floating-point data-processing operation counted by FP_BF16_SPEC due to an Advanced SIMD.

`0x8056`, **SVE_SPEC, Operation speculatively executed, SVE data processing**

The counter counts each operation counted by ASE_SVE_SPEC that is an SVE data-processing operation.

An SVE data-processing operation is any operation due to an instruction that operates on the SVE scalable vector and predicate registers that is not any of the following:

- A load or store operation. These are counted by SVE_INST_SPEC.
- If the Cryptographic Extension is implemented, an operation counted by CRYPTO_SPEC as a Cryptographic data-processing operation.

If the Cryptographic Extension and FEAT_SVE2 are implemented, operations due to the SVE `PMULLB` and `PMULLT` (Q variants) instructions are not counted by CRYPTO_SPEC and are counted as SVE data-processing operations.

`0x8057`, **ASE_SVE_SPEC, Operation speculatively executed, Advanced SIMD data processing or SVE data processing**

The counter counts each operation counted by INST_SPEC that is an Advanced SIMD data-processing operation or an SVE data-processing operation.

See ASE_SPEC and SVE_SPEC for these classifications.

`0x8058`, **NONFP_SPEC, Non-floating-point operation speculatively executed**

The counter counts each speculatively executed data-processing operation due to any of:

- A scalar instruction that would be counted by the DP_SPEC event.
- An A64 Advanced SIMD data processing instruction defined in the section Data processing - SIMD and floating-point that would not be counted by FP_SPEC.
- An SVE instruction with vector source or destination registers that would not be counted by FP_SPEC.

`0x8059`, **ASE_NONFP_SPEC, Non-floating-point operation speculatively executed, Advanced SIMD**

The counter counts each speculatively executed non-floating-point operation counted by ASE_SVE_NONFP_SPEC due to an A64 Advanced SIMD instruction.

`0x805B`, **ASE_SVE_NONFP_SPEC, Non-floating-point operation speculatively executed, Advanced SIMD or SVE**

The counter counts each speculatively executed non-floating-point operation counted by NONFP_SPEC due to an A64 Advanced SIMD instruction.

`0x805D`, **ASE_INT_VREDUCE_SPEC, Integer operation speculatively executed, Advanced SIMD reduction**

The counter counts each speculatively executed across-vector and pairwise integer reduction operation counted by ASE_SVE_INT_VREDUCE_SPEC due to any of the following A64 instructions:

- Advanced SIMD: ADDP, ADDV, SADALP, SADDLP, SADDLV, SMAXP, SMAXV, SMINP, SMINV, UADALP, UADDLP, UADDLV, UMAXP, UMAXV, UMINP, or UMINV.

`0x805F`, **ASE_SVE_INT_VREDUCE_SPEC, Integer operation speculatively executed, Advanced SIMD or SVE reduction**

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The counter counts each speculatively executed across-vector and pairwise integer reduction operation counted by INT_SPEC due to any of the following A64 instructions:

- Advanced SIMD: ADDP, ADDV, SADALP, SADDLP, SADDLV, SMAXP, SMAXV, SMINP, SMINV, UADALP, UADDLP, UADDLV, UMAXP, UMAXV, UMINP, or UMINV.

`0x8084`, **ASE_SVE_LDST_SPEC, Operation speculatively executed, Advanced SIMD or SVE load or store**

The counter counts each speculatively executed operation that reads from or writes to memory due to an A64 Advanced SIMD load, store, or prefetch instruction.

`0x8085`, **ASE_SVE_LD_SPEC, Operation speculatively executed, Advanced SIMD or SVE load**

The counter counts each speculatively executed operation that reads from memory counted by ASE_SVE_LDST_SPEC due to an A64 Advanced SIMD load instruction.

`0x8086`, **ASE_SVE_ST_SPEC, Operation speculatively executed, Advanced SIMD or SVE store**

The counter counts each speculatively executed operation that writes to memory counted by ASE_SVE_LDST_SPEC due to an A64 Advanced SIMD store instruction.

`0x8087`, **PRF_SPEC, Operation speculatively executed, prefetch**

The counter counts each speculatively executed prefetch operation due to any of the following A64 instructions:

- Scalar: PRFM, PRFUM, or RPRFM.

`0x8088`, **BASE_LDST_REG_SPEC, Operation speculatively executed, general-purpose register load, store, or prefetch**

The counter counts each speculatively executed operation that reads from or writes to memory due to a general-purpose register load, store, or prefetch instruction.

`0x8089`, **BASE_LD_REG_SPEC, Operation speculatively executed, general-purpose register load**

The counter counts each speculatively executed operation that reads from memory counted by BASE_LDST_REG_SPEC due to a general-purpose register load instruction.

`0x808A`, **BASE_ST_REG_SPEC, Operation speculatively executed, general-purpose register store**

The counter counts each speculatively executed operation that writes to memory counted by BASE_LDST_REG_SPEC due to a general-purpose register store instruction.

It is IMPLEMENTATION DEFINED whether the counter counts operations due to `DC ZVA` instructions.

`0x808B`, **BASE_PRF_SPEC, Operation speculatively executed, general-purpose register prefetch**

The counter counts each speculatively executed prefetch operation counted by BASE_LDST_REG_SPEC due to any of the following A64 instructions:

- Scalar: PRFM, PRFUM, or RPRFM.

`0x808C`, **FPASE_LDST_REG_SPEC, Operation speculatively executed, SIMD&FP register load or store**

The counter counts each speculatively executed operation that reads from or writes to memory due to any of the following A64 instructions:

- Scalar: LDP (SIMD&FP), LDR (SIMD&FP), STP (SIMD&FP), or STR (SIMD&FP).
- Advanced SIMD: LD1, LD1R, LDNP, LDP, LDR, LDUR, ST1, STNP, STP, STR, or STUR.

`0x808D`, **FPASE_LD_REG_SPEC, Operation speculatively executed, SIMD&FP register load**

The counter counts each speculatively executed operation that reads from memory counted by FPASE_LDST_REG_SPEC due to any of the following A64 instructions:

- Scalar: LDP (SIMD&FP) or LDR (SIMD&FP).

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Advanced SIMD: LD1, LD1R, LDNP, LDP, LDR, or LDUR.

`0x808E`, **FPASE_ST_REG_SPEC, Operation speculatively executed, SIMD&FP register store**

The counter counts each speculatively executed operation that writes to memory counted by FPASE_LDST_REG_SPEC due to any of the following A64 instructions:

- Scalar: STP (SIMD&FP) or STR (SIMD&FP).
- Advanced SIMD: ST1, STNP, STP, STR, or STUR.

`0x80A4`, **ASE_SVE_LDST_MULTI_SPEC, Operation speculatively executed, Advanced SIMD or SVE contiguous load or store multiple vector**

The counter counts each speculatively executed operation due to an A64 Advanced SIMD multiple vector contiguous structure load or store instruction.

`0x80A5`, **ASE_SVE_LD_MULTI_SPEC, Operation speculatively executed, Advanced SIMD or SVE contiguous load multiple vector**

The counter counts each speculatively executed operation that reads from memory counted by ASE_SVE_LDST_MULTI_SPEC due to an A64 Advanced SIMD multiple vector contiguous structure load instruction.

`0x80A6`, **ASE_SVE_ST_MULTI_SPEC, Operation speculatively executed, Advanced SIMD or SVE contiguous store multiple vector**

The counter counts each speculatively executed operation that writes to memory counted by ASE_SVE_LDST_MULTI_SPEC due to an A64 Advanced SIMD multiple vector contiguous structure store instruction.

`0x80B4`, **ASE_SVE_UNALIGNED_LDST_SPEC, Advanced SIMD or SVE unaligned access**

The counter counts each speculatively executed A64 Advanced SIMD load or store instruction where either:

- A contiguous vector address is not aligned to the minimum of the in-memory size of the vector and the cache line size, in bytes.
- A gather, scatter, or single element address is not aligned to the memory element access size, in bytes.

The unaligned access is counted even if it is subsequently converted into multiple aligned accesses.

`0x80B5`, **ASE_SVE_UNALIGNED_LD_SPEC, Advanced SIMD or SVE unaligned read**

The counter counts each unaligned memory access counted by ASE_SVE_UNALIGNED_LDST_SPEC due to an A64 Advanced SIMD load instruction.

The unaligned access is counted even if it is subsequently converted into multiple aligned accesses.

`0x80B6`, **ASE_SVE_UNALIGNED_ST_SPEC, Advanced SIMD or SVE unaligned write**

The counter counts each unaligned memory access counted by ASE_SVE_UNALIGNED_LDST_SPEC due to an A64 Advanced SIMD store instruction.

The unaligned access is counted even if it is subsequently converted into multiple aligned accesses.

`0x80B8`, **ASE_SVE_UNALIGNED_CONTIG_LDST_SPEC, Advanced SIMD or SVE unaligned contiguous access**

The counter counts each speculatively executed Advanced SIMD contiguous load or store operation where the address is not aligned to the minimum of the in-memory size of the vector and the cache line size, in bytes.

The unaligned access is counted even if it is subsequently converted into multiple aligned accesses.

`0x80B9`, **ASE_SVE_UNALIGNED_CONTIG_LD_SPEC, Advanced SIMD or SVE unaligned contiguous read**

The counter counts each unaligned memory access counted by ASE_SVE_UNALIGNED_CONTIG_LDST_SPEC due to an A64 Advanced SIMD load instruction.

The unaligned access is counted even if it is subsequently converted into multiple aligned accesses.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x80BA`, **ASE_SVE_UNALIGNED_CONTIG_ST_SPEC, Advanced SIMD or SVE unaligned contiguous write**

The counter counts each unaligned memory access counted by ASE_SVE_UNALIGNED_CONTIG_LDST_SPEC due to a A64 Advanced SIMD store instruction.

The unaligned access is counted even if it is subsequently converted into multiple aligned accesses.

`0x80C1`, **FP_FIXED_OPS_SPEC, Non-scalable element arithmetic operations speculatively executed, floating-point**

The counter increments by v for each speculatively executed non-scalable element arithmetic operation, due to an instruction where the type was floating-point.

Where v is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

The counter does not count operations that are counted by FP_SCALE_OPS_SPEC.

`0x80C3`, **FP_HP_FIXED_OPS_SPEC, Non-scalable element arithmetic operations speculatively executed, largest type is half-precision floating-point**

The counter increments by v for each speculatively executed non-scalable element arithmetic operation, due to an instruction where the largest type was half-precision floating-point.

Where v is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

The counter does not count operations that are counted by FP_SCALE_OPS_SPEC.

`0x80C5`, **FP_SP_FIXED_OPS_SPEC, Non-scalable element arithmetic operations speculatively executed, largest type is single-precision floating-point**

The counter increments by v for each speculatively executed non-scalable element arithmetic operation, due to an instruction where the largest type was single-precision floating-point.

Where v is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

The counter does not count operations that are counted by FP_SCALE_OPS_SPEC.

`0x80C7`, **FP_DP_FIXED_OPS_SPEC, Non-scalable element arithmetic operations speculatively executed, largest type is double-precision floating-point**

The counter increments by v for each speculatively executed non-scalable element arithmetic operation, due to an instruction where the largest type was double-precision floating-point.

Where v is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

The counter does not count operations that are counted by FP_SCALE_OPS_SPEC.

`0x80C9`, **INT_FIXED_OPS_SPEC, Non-scalable element arithmetic operations speculatively executed, integer**

The counter increments by v for each speculatively executed non-scalable element arithmetic operation, due to an instruction where the type was integer.

Where v is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

The counter does not count operations that are counted by FP_SCALE_OPS_SPEC.

`0x80CB`, **LDST_FIXED_OPS_SPEC, Non-scalable load or store element operations speculatively executed**

The counter counts each speculatively executed Memory-read operation or Memory-write operation due to any of:

- Any load, store, or atomic operation, other than loads and stores of the SVE P and Z registers.
- Any SVE non-vector load or store operation.
- An SVE load and replicate instruction, including: `LD1R`, `LD1RB`, `LD1RD`, `LD1RH`, `LD1RQ`, `LD1RQB`, `LD1RQD`, `LD1RQH`, `LD1RQW`, `LD1RSB`, `LD1RSH`, or `LD1RSW`.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For each instruction, the counter is incremented by the number of operations specified by the instruction. For example, the counter counts operations as follows:

- Non-SVE load and store of a single register instructions increment the counter by 1. This includes loads and stores of Sx, Dx, and Qx SIMD&FP registers.
- Non-SVE load and store of a pair of registers instructions increment the counter by 2. This includes loads and stores of Sx, Dx, and Qx SIMD&FP registers.
- Atomic store instructions increment the counter by 1. These are instructions that atomically update a value in memory without returning a value to a register.
- Atomic load, compare and swap of a single register, and swap instructions increment the counter by 2. Atomic load instructions are instructions that atomically update a value in memory, returning a value to a register.
- Compare and swap of a pair of registers increment the counter by 4.
- Advanced SIMD `LD1R` instructions increment the counter by 1.
- Advanced SIMD `LD[1-4]` and `ST[1-4]` instructions increment the counter by the number of elements transferred per vector multiplied by the number of transferred registers.
- `DC ZVA` and `DC GZVA` instructions increment by an IMPLEMENTATION DEFINED amount.

Note

When a replicating load instruction loads data of the same size as the vector length, which causes no replication to occur, the instruction is counted.

`0x80CD`**, LD_FIXED_OPS_SPEC, Non-scalable load element operations speculatively executed**

The counter counts each speculatively executed Memory-read operation counted by LDST_FIXED_OPS_SPEC.

For each instruction, the counter is incremented by the number of operations specified by the instruction. That is, the counter is incremented by:

- Half the value that the LDST_FIXED_OPS_SPEC event counts if the operation is a load atomic, compare and swap, or swap operation.
- The same as for LDST_FIXED_OPS_SPEC if the operation is any other load operation.

`0x80CF`**, ST_FIXED_OPS_SPEC, Non-scalable store element operations speculatively executed**

The counter counts each Memory-write operation counted by LDST_FIXED_OPS_SPEC.

For each instruction, the counter is incremented by the number of operations specified by the instruction. That is, the counter is incremented by:

- Half the value that the LDST_FIXED_OPS_SPEC event counts if the operation is a store atomic, compare and swap, or swap operation.
- The same as for LDST_FIXED_OPS_SPEC if the operation is any other store operation.

`0x80DB`**, LDST_FIXED_BYTES_SPEC, Non-scalable load and store bytes speculatively executed**

The counter counts each byte speculatively read or written due to any of:

- Any load, store, or atomic operation, other than loads and stores of the SVE P and Z registers.
- Any SVE non-vector load or store operation.
- An SVE load and replicate instruction, including: `LD1R`, `LD1RB`, `LD1RD`, `LD1RH`, `LD1RQ`, `LD1RQB`, `LD1RQD`, `LD1RQH`, `LD1RQW`, `LD1RSB`, `LD1RSH`, or `LD1RSW`.

For each instruction, the counter is incremented by the number of bytes transferred per register multiplied by the number of registers transferred multiplied by the number of transfers made per register. For example, the counter counts bytes as follows:

- Non-SVE load and store of a single register instructions increment the counter by (MSIZE ÷ 8). This includes loads and stores of Sx, Dx, and Qx SIMD&FP registers.
- Non-SVE load and store of a pair of registers instructions increment the counter by 2 × (MSIZE ÷ 8). This includes loads and stores of Sx, Dx, and Qx SIMD&FP registers.
- Atomic store instructions increment the counter by (MSIZE ÷ 8). These are instructions that atomically update a value in memory without returning a value to a register.
- Atomic load, compare and swap of a single register, and swap instructions increment the counter by 2 × (MSIZE ÷ 8). Atomic load instructions are instructions that atomically update a value in memory, returning a value to a register.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Compare and swap of a pair of registers increment the counter by 4 × (MSIZE ÷ 8).
- Advanced SIMD `LD1R` instructions increment the counter by (MSIZE ÷ 8).
- Advanced SIMD `LD[1-4]` and `ST[1-4]` instructions increment the counter by the number of registers transferred multiplied by the number of bytes being transferred per register.
- `DC ZVA` instructions increment by the counter by $2^{(DCZID_EL0.BS)}$.

`0x80DD`, **LD_FIXED_BYTES_SPEC, Non-scalable load bytes speculatively executed**

The counter counts each speculatively read byte counted by LDST_FIXED_BYTES_SPEC.

For each instruction, the counter is incremented by the number of bytes transferred per register multiplied by the number of registers transferred. That is, the counter is incremented by:

- Half the value that the LDST_FIXED_BYTES_SPEC event counts if the operation is a load atomic, compare and swap, or swap operation.
- The same as for LDST_FIXED_BYTES_SPEC if the operation is any other load operation.

`0x80DF`, **ST_FIXED_BYTES_SPEC, Non-scalable store bytes speculatively executed**

The counter counts each speculatively written byte counted by LDST_FIXED_BYTES_SPEC.

For each instruction, the counter is incremented by the number of bytes transferred per register multiplied by the number of registers transferred. That is, the counter is incremented by:

- Half the value that the LDST_FIXED_BYTES_SPEC event counts if the operation is a swap, or compare and swap operation.
- The same as for LDST_FIXED_BYTES_SPEC if the operation is any other store operation, including an atomic store operation.

`0x80E1`, **ASE_INT8_SPEC, Integer operation speculatively executed, Advanced SIMD 8-bit**

The counter counts each operation counted by ASE_SVE_INT8_SPEC that is an Advanced SIMD operation.

`0x80E3`, **ASE_SVE_INT8_SPEC, Integer operation speculatively executed, Advanced SIMD or SVE 8-bit**

The counter counts each operation counted by ASE_SVE_INT_SPEC where the largest type is 8-bit integer.

`0x80E5`, **ASE_INT16_SPEC, Integer operation speculatively executed, Advanced SIMD 16-bit**

The counter counts each operation counted by ASE_SVE_INT16_SPEC that is an Advanced SIMD operation.

`0x80E7`, **ASE_SVE_INT16_SPEC, Integer operation speculatively executed, Advanced SIMD or SVE 16-bit**

The counter counts each operation counted by ASE_SVE_INT_SPEC where the largest type is 16-bit integer.

`0x80E9`, **ASE_INT32_SPEC, Integer operation speculatively executed, Advanced SIMD 32-bit**

The counter counts each operation counted by ASE_SVE_INT32_SPEC that is an Advanced SIMD operation.

`0x80EB`, **ASE_SVE_INT32_SPEC, Integer operation speculatively executed, Advanced SIMD or SVE 32-bit**

The counter counts each operation counted by ASE_SVE_INT_SPEC where the largest type is 32-bit integer.

`0x80ED`, **ASE_INT64_SPEC, Integer operation speculatively executed, Advanced SIMD 64-bit**

The counter counts each operation counted by ASE_SVE_INT64_SPEC that is an Advanced SIMD operation.

`0x80EF`, **ASE_SVE_INT64_SPEC, Integer operation speculatively executed, Advanced SIMD or SVE 64-bit**

The counter counts each operation counted by ASE_SVE_INT_SPEC where the largest type is 64-bit integer.

`0x80F1`, **ASE_FP_DOT_SPEC, Floating-point operation speculatively executed, Advanced SIMD dot-product**

The counter counts each dot-product operation counted by ASE_SVE_FP_DOT_SPEC due to any of the following A64 instructions:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x80F3`, **ASE_SVE_FP_DOT_SPEC, Floating-point operation speculatively executed, Advanced SIMD or SVE dot-product**

The counter counts each dot-product operation counted by FP_SPEC due to any of the following A64 instructions:

`0x80F5`, **ASE_FP_MMLA_SPEC, Floating-point operation speculatively executed, Advanced SIMD matrix multiply**

The counter counts each floating-point matrix multiply operation counted by ASE_SVE_FP_MMLA_SPEC due to any of the following A64 instructions:

`0x80F7`, **ASE_SVE_FP_MMLA_SPEC, Floating-point operation speculatively executed, Advanced SIMD or SVE matrix multiply**

The counter counts each floating-point matrix multiply operation counted by FP_SPEC due to any of the following A64 instructions:

`0x80F9`, **ASE_INT_DOT_SPEC, Operation speculatively executed, Advanced SIMD integer dot-product**

The counter counts each integer dot product operation counted by ASE_SVE_INT_DOT_SPEC due to any of the following A64 instructions:

- When FEAT_DotProd is implemented, Advanced SIMD: SDOT or UDOT.

`0x80FB`, **ASE_SVE_INT_DOT_SPEC, Integer operation speculatively executed, Advanced SIMD or SVE dot-product**

The counter counts each integer dot product operation counted by INT_SPEC due to any of the following A64 instructions:

- When FEAT_DotProd is implemented, Advanced SIMD: SDOT or UDOT.

`0x80FD`, **ASE_INT_MMLA_SPEC, Integer operation speculatively executed, Advanced SIMD matrix multiply**

The counter counts each integer matrix multiply operation counted by ASE_SVE_INT_MMLA_SPEC due to any of the following A64 instructions:

`0x80FF`, **ASE_SVE_INT_MMLA_SPEC, Integer operation speculatively executed, Advanced SIMD or SVE matrix multiply**

The counter counts each integer matrix multiply operation counted by INT_SPEC due to any of the following A64 instructions:

`0x8110`, **BR_IMMED_PRED_RETIRED, Branch instruction architecturally executed, predicted immediate**

The counter counts each instruction counted by both BR_IMMED_RETIRED and BR_PRED_RETIRED.

These are all direct branch instructions on the architecturally executed path, where the branch is correctly predicted.

`0x8111`, **BR_IMMED_MIS_PRED_RETIRED, Branch instruction architecturally executed, mispredicted immediate**

The counter counts each instruction counted by both BR_IMMED_RETIRED and BR_MIS_PRED_RETIRED.

These are all direct branch instructions on the architecturally executed path, where the branch is mispredicted.

`0x8112`, **BR_IND_PRED_RETIRED, Branch instruction architecturally executed, predicted indirect**

The counter counts each instruction counted by both BR_IND_RETIRED and BR_PRED_RETIRED.

These are all indirect branch instructions, including return instructions, on the architecturally executed path, where the branch is correctly predicted.

`0x8113`, **BR_IND_MIS_PRED_RETIRED, Branch instruction architecturally executed, mispredicted indirect**

The counter counts each instruction counted by both BR_IND_RETIRED and BR_MIS_PRED_RETIRED.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

These are all indirect branch instructions, including return instructions, on the architecturally executed path, where the branch is mispredicted.

`0x8114`, **BR_RETURN_PRED_RETIRED, Branch instruction architecturally executed, predicted procedure return**

The counter counts each instruction counted by BR_IND_PRED_RETIRED where if taken, the branch would be counted by BR_RETURN_RETIRED.

These are all return instructions on the architecturally executed path, where the branch is correctly predicted.

`0x8115`, **BR_RETURN_MIS_PRED_RETIRED, Branch instruction architecturally executed, mispredicted procedure return**

The counter counts each instruction counted by BR_IND_MIS_PRED_RETIRED where if taken, the branch would also be counted by BR_RETURN_RETIRED.

These are all return instructions on the architecturally executed path, where the branch is mispredicted.

`0x8116`, **BR_INDNR_PRED_RETIRED, Branch instruction architecturally executed, predicted indirect excluding procedure return**

The counter counts each instruction counted by BR_IND_PRED_RETIRED where if taken, the branch would not be counted by BR_RETURN_RETIRED.

These are all indirect branch instructions, excluding return instructions, on the architecturally executed path, where the branch is correctly predicted.

`0x8117`, **BR_INDNR_MIS_PRED_RETIRED, Branch instruction architecturally executed, mispredicted indirect excluding procedure return**

The counter counts each instruction counted by BR_IND_MIS_PRED_RETIRED where if taken, the branch would not be counted by BR_RETURN_RETIRED.

These are all indirect branch instructions, excluding return instructions, on the architecturally executed path, where the branch is mispredicted.

`0x8118`, **BR_TAKEN_PRED_RETIRED, Branch instruction architecturally executed, predicted branch, taken**

The counter counts each instruction counted by both BR_PRED_RETIRED and PC_WRITE_RETIRED.

These are all branch instructions on the architecturally executed path, where the branch is correctly predicted and taken.

`0x8119`, **BR_TAKEN_MIS_PRED_RETIRED, Branch instruction architecturally executed, mispredicted branch, taken**

The counter counts each instruction counted by both BR_MIS_PRED_RETIRED and PC_WRITE_RETIRED.

These are all branch instructions on the architecturally executed path, where the branch is mispredicted and taken.

`0x811A`, **BR_SKIP_PRED_RETIRED, Branch instruction architecturally executed, predicted branch, not taken**

The counter counts each instruction counted by both BR_PRED_RETIRED and BR_SKIP_RETIRED.

These are all branch instructions on the architecturally executed path, where the branch is correctly predicted and not taken.

`0x811B`, **BR_SKIP_MIS_PRED_RETIRED, Branch instruction architecturally executed, mispredicted branch, not taken**

The counter counts each instruction counted by both BR_MIS_PRED_RETIRED and BR_SKIP_RETIRED.

These are all branch instructions on the architecturally executed path, where the branch is mispredicted and not taken.

`0x811C`, **BR_PRED_RETIRED, Branch instruction architecturally executed, predicted branch**

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The counter counts each instruction counted by BR_RETIRED that is not counted by BR_MIS_PRED_RETIRED.

These are all branch instructions on the architecturally executed path, where the branch is correctly predicted.

`0x8120`, **INST_FETCH_PERCYC, Instruction fetches in progress**

The counter increments by the number of instruction fetches counted by INST_FETCH in progress on each Processor cycle.

The ratio INST_FETCH_PERCYC ÷ INST_FETCH is the mean duration of instruction fetches in Processor cycles.

`0x8121`, **MEM_ACCESS_RD_PERCYC, Data memory reads in progress**

The counter increments by the number of data memory reads counted by MEM_ACCESS_RD in progress on each Processor cycle.

The ratio MEM_ACCESS_RD_PERCYC ÷ MEM_ACCESS_RD is the mean duration of data memory reads in Processor cycles.

`0x8122`, **SAMPLE_FEED_DS, Statistical Profiling sample taken, selected Data Source**

The counter counts each SPE sample that is a load operations where PMSDSFR_EL1[s] is 1 and s is bits [5:0] of the sampled Data Source.

The values of PMSFCR_EL1.FDS are ignored when generating this event.

Samples that are removed by filtering, or discarded, and not written to the Profiling Buffer are counted.

`0x8123`, **SAMPLE_BUFFER_FULL, Profiling Buffer full**

The counter counts each Profiling Buffer full management event.

`0x8124`, **INST_FETCH, Instruction memory access**

The counter counts each Instruction memory access.

The counter increments whether the access results in an access to a Level 1 instruction cache, a Level 2 instruction, data or unified cache, or none of these.

The counter does not increment as a result of:

- Data memory accesses.
- Translation table walks.
- Refilling of any cache.
- Accesses that result from cache maintenance instructions.

`0x8125`, **BUS_REQ_RD_PERCYC, Bus read transactions in progress**

The counter increments by the number of bus read transactions counted by BUS_REQ_RD in progress on each Processor cycle.

The ratio BUS_REQ_RD_PERCYC ÷ BUS_REQ_RD is the mean duration of bus read transactions in Processor cycles.

A bus read transaction is in progress between the transaction starting and it completing. This includes cycles when the transaction is not occupying the bus.

If the bus is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the bus is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x8126`, **BUS_REQ_WR_PERCYC, Bus write transactions in progress**

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The counter increments by the number of bus write transactions counted by BUS_REQ_WR in progress on each Processor cycle.

The ratio BUS_REQ_WR_PERCYC ÷ BUS_REQ_WR is the mean duration of bus write transactions in Processor cycles.

A bus write transaction is in progress between the transaction starting and it completing. This includes cycles when the transaction is not occupying the bus.

A mean duration value calculated using this event only measures the latency from the perspective of the bus. This is not necessarily the same as the latency for the write to be Complete, as the write might be completed on the bus before reaching its endpoint.

For some bus implementations, writes are posted, meaning that, from the perspective of the bus, the write completes immediately. There is no response when the write reaches its Completion endpoint. This event is not included for such implementations.

If the bus is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the bus is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x8127`**, PMU_SNAPSHOT, Successful PMU capture event**

The counter counts each PMU snapshot Capture event that was successful, that is, PMSSCR_EL1.NC is set to 0.

It is CONSTRAINED UNPREDICTABLE whether the counter counts successful Capture events when the PE is in Debug state.

`0x8128`**, DTLB_WALK_PERCYC, Data translation table walks in progress**

The counter increments by the number of data translation table walks counted by DTLB_WALK in progress on each Processor cycle.

The ratio DTLB_WALK_PERCYC ÷ DTLB_WALK is the mean duration of data translation table walks in Processor cycles.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x8129`**, ITLB_WALK_PERCYC, Instruction translation table walks in progress**

The counter increments by the number of instruction translation table walks counted by ITLB_WALK in progress on each Processor cycle.

The ratio ITLB_WALK_PERCYC ÷ ITLB_WALK is the mean duration of instruction translation table walks in Processor cycles.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x8130`**, L1D_TLB_RW, Level 1 data TLB access, demand access**

The counter counts each access counted by L1D_TLB that is due to a demand Memory-read operation or demand Memory-write operation.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

Arm recommends that this event is implemented if event L1D_TLB_PRFM is implemented.

`0x8131`**, L1I_TLB_RD, Level 1 instruction TLB access, demand access**

The counter counts each access counted by L1I_TLB that is due to a demand Instruction memory access.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

Arm recommends that this event is implemented if event L1I_TLB_PRFM is implemented.

`0x8132`**, L1D_TLB_PRFM, Level 1 data TLB access, software prefetch**

The counter counts each access counted by L1D_TLB_PRF that is due to a prefetch instruction.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

Arm recommends that this event is implemented if event L1D_TLB_RW is implemented.

`0x8133`**, L1I_TLB_PRFM, Level 1 instruction TLB access, software prefetch**

The counter counts each access counted by L1I_TLB_PRF that is due to a prefetch instruction.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

Arm recommends that this event is implemented if event L1I_TLB_RD is implemented.

`0x8134`**, DTLB_HWUPD, Data TLB hardware update of translation table**

The counter counts each access counted by L1D_TLB that causes a hardware update of a translation table entry.

Each attempted hardware update of a translation table entry is counted once. If the PE requires multiple translation table walk accesses to perform an update, this counts as a single update. If the update fails because it would not be atomic and has to be retried, each retry is counted.

The counter does not count the access if any of the following are true:

- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- The access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the access if the access generates a Translation fault for any other reason.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x8135`**, ITLB_HWUPD, Instruction TLB hardware update of translation table**

The counter counts each access counted by L1I_TLB that causes a hardware update of a translation table entry.

Each attempted hardware update of a translation table entry is counted once. If the PE requires multiple translation table walk accesses to perform an update, this counts as a single update. If the update fails because it would not be atomic and has to be retried, each retry is counted.

The counter does not count the access if any of the following are true:

- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- The access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the access if the access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x8136`**, DTLB_STEP, Data TLB translation table walk, step**

The counter counts each translation table walk access due to a refill of the data or unified TLB.

The event is Attributable to the access that missed in the TLB and caused the walk, not to the owner of the translation tables being accessed. For example, this means that if an EL0 access causes a translation table walk consisting of accesses to both stage 1 and stage 2 translation tables, all accesses are counted if event counting is allowed at EL0, regardless of whether event counting is allowed at EL1 and/or EL2.

The counter does not count the translation table walk access if any of the following are true:

- The access causing the refill generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- The access causing the refill is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the translation table walk access if the access causing the refill generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x8137`**, ITLB_STEP, Instruction TLB translation table walk, step**

The counter counts each translation table walk access due to a refill of the instruction TLB.

The event is Attributable to the access that missed in the TLB and caused the walk, not to the owner of the translation tables being accessed. For example, this means that if an EL0 access causes a translation table walk consisting of accesses to both stage 1 and stage 2 translation tables, all accesses are counted if event counting is allowed at EL0, regardless of whether event counting is allowed at EL1 and/or EL2.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The counter does not count the translation table walk access if any of the following are true:

- The access causing the refill generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- The access causing the refill is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the translation table walk access if the access causing the refill generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x8138`, DTLB_WALK_LARGE, Data TLB large page translation table walk

The counter counts each translation table walk counted by DTLB_WALK where the result of the walk yields a large page size.

The set of large page sizes is the complement of the set of small page sizes defined by the DTLB_WALK_SMALL event. For example, these translations might cached by dedicated TLB resources. This set is IMPLEMENTATION DEFINED and might differ between instruction and data TLBs.

The counter does not count the walk if the access generates a Translation fault.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x8139`, ITLB_WALK_LARGE, Instruction TLB large page translation table walk

The counter counts each translation table walk counted by ITLB_WALK where the result of the walk yields a large page size.

The set of large page sizes is the complement of the set of small page sizes defined by the ITLB_WALK_SMALL event. For example, these translations might cached by dedicated TLB resources. This set is IMPLEMENTATION DEFINED and might differ between instruction and data TLBs.

The counter does not count the walk if the access generates a Translation fault.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x813A`, DTLB_WALK_SMALL, Data TLB small page translation table walk

The counter counts each translation table walk counted by DTLB_WALK where the result of the walk yields a small page size.

The set of small page sizes is the complement of the set of large page sizes defined by the DTLB_WALK_LARGE event. For example, these translations might cached by dedicated TLB resources. This set is IMPLEMENTATION DEFINED and might differ between instruction and data TLBs.

The counter does not count the walk if the access generates a Translation fault.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x813B`**, ITLB_WALK_SMALL, Instruction TLB small page translation table walk**

The counter counts each translation table walk counted by ITLB_WALK where the result of the walk yields a small page size.

The set of small page sizes is the complement of the set of large page sizes defined by the ITLB_WALK_LARGE event. For example, these translations might cached by dedicated TLB resources. This set is IMPLEMENTATION DEFINED and might differ between instruction and data TLBs.

The counter does not count the walk if the access generates a Translation fault.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x813C`**, DTLB_WALK_RW, Data TLB demand access, with at least one translation table walk**

The counter counts each demand access counted by L1D_TLB_RW that causes a refill or update of a data or unified TLB involving at least one translation table walk access.

The counter does not count the demand access if any of the following are true:

- The access is due to a TLB maintenance instruction.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the demand access if the access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x813D`**, ITLB_WALK_RD, Instruction TLB demand access, with at least one translation table walk**

The counter counts each demand access counted by L1I_TLB_RD that causes a refill or update of an instruction TLB involving at least one translation table walk access.

The counter does not count the demand access if any of the following are true:

- The access is due to a TLB maintenance instruction.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the demand access if the access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x813E`, **DTLB_WALK_PRFM, Data TLB software prefetch, with at least one translation table walk**

The counter counts each software prefetch access counted by L1D_TLB_PRFM that causes a refill or update of a data or unified TLB involving at least one translation table walk access.

The counter does not count the software prefetch access if any of the following are true:

- The access is due to a TLB maintenance instruction.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the software prefetch access if the access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x813F`, **ITLB_WALK_PRFM, Instruction TLB software prefetch, with at least one translation table walk**

The counter counts each software prefetch access counted by L1I_TLB_PRFM that causes a refill or update of an instruction TLB involving at least one translation table walk access.

The counter does not count the software prefetch access if any of the following are true:

- The access is due to a TLB maintenance instruction.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the software prefetch access if the access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x8140`, **L1D_CACHE_RW, Level 1 data cache demand access**

The counter counts each access counted by L1D_CACHE that is due to a demand Memory-read operation or demand Memory-write operation.

This includes accesses made by speculatively executed instructions.

Arm recommends that this event is implemented if any of the following are true:

- Event L1D_CACHE_PRFM is implemented.
- Event L1D_CACHE_HWPRF is implemented.

`0x8141`, **L1I_CACHE_RD, Level 1 instruction cache demand fetch**

The counter counts each access counted by L1I_CACHE that is due to a demand Instruction memory access.

This includes instruction prefetches made by the PE for speculatively executed instructions.

Arm recommends that this event is implemented if any of the following are true:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Event L1I_CACHE_PRFM is implemented.
- Event L1I_CACHE_HWPRF is implemented.

`0x8142`, **L1D_CACHE_PRFM, Level 1 data cache software prefetch**

The counter counts each access counted by L1D_CACHE_PRF that is due to a prefetch instruction.

Arm recommends that this event is implemented if event L1D_CACHE_RW is implemented.

`0x8143`, **L1I_CACHE_PRFM, Level 1 instruction cache software prefetch**

The counter counts each access counted by L1I_CACHE_PRF that is due to a prefetch instruction.

Arm recommends that this event is implemented if event L1I_CACHE_RD is implemented.

`0x8144`, **L1D_CACHE_MISS, Level 1 data cache demand access miss**

The counter counts each demand access counted by L1D_CACHE_RW that misses in the Level 1 data or unified cache, causing an access to outside of the Level 1 caches of this PE.

`0x8145`, **L1I_CACHE_HWPRF, Level 1 instruction cache hardware prefetch**

The counter counts each access counted by L1I_CACHE_PRF that is due to a hardware prefetch.

The hardware prefetch is generated by a hardware prefetcher at the Level 1 instruction or unified cache.

The PE might prefetch instructions as part of instruction pipelining, and might do so for speculatively executed instruction paths. PE instruction prefetching is not counted as hardware prefetching. Unlike PE instruction prefetching, a hardware prefetch only allocates instructions into the cache.

`0x8146`, **L1D_CACHE_REFILL_PRFM, Level 1 data cache refill, software prefetch**

The counter counts each access counted by L1D_CACHE_PRFM that causes a refill of the Level 1 data or unified cache from outside of the Level 1 data or unified cache.

`0x8147`, **L1I_CACHE_REFILL_PRFM, Level 1 instruction cache refill, software prefetch**

The counter counts each access counted by L1I_CACHE_PRFM that causes a refill of the Level 1 instruction or unified cache from outside of the Level 1 instruction or unified cache.

`0x8148`, **L2D_CACHE_RW, Level 2 data cache demand access**

The counter counts each access counted by L2D_CACHE that is due to a demand Memory-read operation or demand Memory-write operation.

This includes:

- Accesses made by speculatively executed instructions.
- Accesses to the Level 2 data or unified cache due to a refill of another cache caused by a demand Memory-read operation or demand Memory-write operation.

Arm recommends that this event is implemented if any of the following are true:

- Event L2D_CACHE_PRFM is implemented.
- Event L2D_CACHE_HWPRF is implemented.

`0x8149`, **L2I_CACHE_RD, Level 2 instruction cache demand fetch**

The counter counts each access counted by L2I_CACHE that is due to a demand Instruction memory access.

This includes:

- Instruction prefetches made by the PE for speculatively executed instructions.
- Accesses to the Level 2 instruction or unified cache due to a refill of another cache caused by a demand Instruction memory access.

Arm recommends that this event is implemented if any of the following are true:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Event L2I_CACHE_PRFM is implemented.
- Event L2I_CACHE_HWPRF is implemented.

`0x814A`, **L2D_CACHE_PRFM, Level 2 data cache software prefetch**

The counter counts each access counted by L2D_CACHE_PRF that is due to a prefetch instruction.

This includes accesses to the Level 2 data or unified cache due to a refill of another cache caused by a prefetch instruction.

Arm recommends that this event is implemented if event L2D_CACHE_RW is implemented.

`0x814B`, **L2I_CACHE_PRFM, Level 2 instruction cache software prefetch**

The counter counts each access counted by L2I_CACHE_PRF that is due to a prefetch instruction.

This includes accesses to the Level 2 instruction or unified cache due to a refill of another cache caused by a prefetch instruction.

Arm recommends that this event is implemented if event L2I_CACHE_RD is implemented.

`0x814C`, **L2D_CACHE_MISS, Level 2 data cache demand access miss**

The counter counts each demand access counted by L2D_CACHE_RW that misses in the Level 1 and Level 2 data or unified caches, causing an access to outside of the Level 1 and Level 2 caches of this PE.

`0x814D`, **L2I_CACHE_HWPRF, Level 2 instruction cache hardware prefetch**

The counter counts each access counted by L2I_CACHE_PRF that is due to a hardware prefetch.

The hardware prefetch is generated by a hardware prefetcher at the Level 2 instruction or unified cache.

`0x814E`, **L2D_CACHE_REFILL_PRFM, Level 2 data cache refill, software prefetch**

The counter counts each access counted by L2D_CACHE_PRFM that causes a refill of the Level 2 data or unified cache, or any Level 1 data, instruction, or unified cache of this PE, from outside of those caches.

`0x814F`, **L2I_CACHE_REFILL_PRFM, Level 2 instruction cache refill, software prefetch**

The counter counts each access counted by L2I_CACHE_PRFM that causes a refill of the Level 2 instruction or unified cache, or any Level 1 data, instruction, or unified cache of this PE, from outside of those caches.

`0x8150`, **L3D_CACHE_RW, Level 3 data cache demand access**

The counter counts each access counted by L3D_CACHE that is due to a demand Memory-read operation or demand Memory-write operation.

This includes:

- Accesses made by speculatively executed instructions.
- Accesses to the Level 3 data or unified cache due to a refill of another cache caused by a demand Memory-read operation or demand Memory-write operation.

Arm recommends that this event is implemented if any of the following are true:

- Event L3D_CACHE_PRFM is implemented.
- Event L3D_CACHE_HWPRF is implemented.

`0x8151`, **L3D_CACHE_PRFM, Level 3 data cache software prefetch**

The counter counts each access counted by L3D_CACHE_PRF that is due to a prefetch instruction.

This includes accesses to the Level 3 data or unified cache due to a refill of another cache caused by a prefetch instruction.

Arm recommends that this event is implemented if event L3D_CACHE_RW is implemented.

`0x8152`, **L3D_CACHE_MISS, Level 3 data cache demand access miss**

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The counter counts each demand access counted by L3D_CACHE_RW that misses in the Level 1 to Level 3 data or unified caches, causing an access to outside of the Level 1 to Level 3 caches of this PE.

`0x8153`, **L3D_CACHE_REFILL_PRFM, Level 3 data cache refill, software prefetch**

The counter counts each access counted by L3D_CACHE_PRFM that causes a refill of the Level 3 data or unified cache, or any Level 1 or Level 2 data, instruction, or unified cache of this PE, from outside of those caches.

`0x8154`, **L1D_CACHE_HWPRF, Level 1 data cache hardware prefetch**

The counter counts each access counted by L1D_CACHE_PRF that is due to a hardware prefetch.

The hardware prefetch is generated by a hardware prefetcher at the Level 1 data or unified cache.

`0x8155`, **L2D_CACHE_HWPRF, Level 2 data cache hardware prefetch**

The counter counts each access counted by L2D_CACHE_PRF that is due to a hardware prefetch.

The hardware prefetch is generated by a hardware prefetcher at the Level 2 data or unified cache.

`0x8156`, **L3D_CACHE_HWPRF, Level 3 data cache hardware prefetch**

The counter counts each access counted by L3D_CACHE_PRF that is due to a hardware prefetch.

The hardware prefetch is generated by a hardware prefetcher at the Level 3 data or unified cache.

`0x8157`, **LL_CACHE_HWPRF, Last level cache hardware prefetch**

The counter counts each access counted by LL_CACHE_PRF that is due to a hardware prefetch.

The hardware prefetch is generated by a hardware prefetcher at the Last level cache.

`0x8158`, **STALL_FRONTEND_MEMBOUND, Frontend stall cycles, memory bound**

The counter counts each cycle counted by STALL_FRONTEND when no instructions are delivered from the memory system.

The counter counts each stall that occurs when the frontend interface to memory is busy or stalled.

This includes the stalls counted by STALL_FRONTEND_L1I, STALL_FRONTEND_L2I, STALL_FRONTEND_MEM, and STALL_FRONTEND_TLB, and any other IMPLEMENTATION DEFINED memory stalls.

It does not include stalls that are counted by STALL_FRONTEND_CPUBOUND. However both events will count the same cycle counted by STALL_FRONTEND if there are both memory and processor-resource stall conditions active.

`0x8159`, **STALL_FRONTEND_L1I, Frontend stall cycles, level 1 instruction cache**

The counter counts each cycle counted by STALL_FRONTEND_MEMBOUND when there is a demand instruction miss in the first level of instruction cache.

The counter does not count the cycle if any of the following are true:

- The STALL_FRONTEND_L2I event is implemented and there is a demand instruction miss in the second level of instruction cache, meaning the STALL_FRONTEND_L2I event counts the cycle.
- There is a demand instruction miss in the last level of instruction cache within the PE clock domain, meaning the STALL_FRONTEND_MEM event counts the cycle.

`0x815A`, **STALL_FRONTEND_L2I, Frontend stall cycles, level 2 instruction cache**

The counter counts each cycle counted by STALL_FRONTEND_MEMBOUND when there is a demand instruction miss in the second level of instruction or unified cache.

The counter does not count the cycle if there is a demand instruction miss in the last level of instruction cache within the PE clock domain, meaning the STALL_FRONTEND_MEM event counts the cycle.

`0x815B`, **STALL_FRONTEND_MEM, Frontend stall cycles, last level PE cache or memory**

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The counter counts each cycle counted by STALL_FRONTEND_MEMBOUND when there is a demand instruction miss in the last level of instruction or unified cache within the PE clock domain or a non-cacheable instruction fetch in progress.

`0x815C`, **STALL_FRONTEND_TLB, Frontend stall cycles, TLB**

The counter counts each cycle counted by STALL_FRONTEND_MEMBOUND when there is a demand instruction miss in the instruction or unified TLB.

`0x8160`, **STALL_FRONTEND_CPUBOUND, Frontend stall cycles, processor bound**

The counter counts each cycle counted by STALL_FRONTEND when the frontend is stalled on a frontend processor resource, not including memory.

The counter counts each stall that occurs when a frontend processor resource is busy.

This includes the stalls counted by STALL_FRONTEND_FLOW, STALL_FRONTEND_FLUSH, and STALL_FRONTEND_RENAME, and any other IMPLEMENTATION DEFINED processor resource stalls.

It does not include stalls that are counted by STALL_FRONTEND_MEMBOUND. However both events will count the same cycle counted by STALL_FRONTEND if there are both memory and processor-resource stall conditions active.

`0x8161`, **STALL_FRONTEND_FLOW, Frontend stall cycles, flow control**

The counter counts each cycle counted by STALL_FRONTEND_CPUBOUND when the frontend is stalled on unavailability of prediction flow resources.

Note

This event is not counting stalls due to mispredictions, but rather stalls when the frontend is unable to make a prediction.

`0x8162`, **STALL_FRONTEND_FLUSH, Frontend stall cycles, flush recovery**

The counter counts each cycle counted by STALL_FRONTEND_CPUBOUND when the frontend is recovering from a pipeline flush.

The situations where the frontend is flushed are IMPLEMENTATION DEFINED. For example, the frontend might be flushed on a branch misprediction or on a Context synchronization event.

`0x8163`, **STALL_FRONTEND_RENAME, Frontend stall cycles, rename full**

The counter counts each cycle counted by STALL_FRONTEND_CPUBOUND when operations are available from the frontend but at least one is not ready to be sent to the backend because no rename register is available.

If this event is implemented and counts such stalls then the STALL_BACKEND_RENAME event counts as zero.

`0x8164`, **STALL_BACKEND_MEMBOUND, Backend stall cycles, memory bound**

The counter counts each cycle counted by STALL_BACKEND when the backend is waiting for a memory access to complete.

The counter counts each stall that occurs when the backend interface to memory is busy or stalled.

This includes the stalls counted by: STALL_BACKEND_ATOMIC, STALL_BACKEND_L1D, STALL_BACKEND_L2D, STALL_BACKEND_MEM, STALL_BACKEND_ST, STALL_BACKEND_TLB, and any other IMPLEMENTATION DEFINED memory stall events.

It does not include stalls that are counted by STALL_BACKEND_CPUBOUND, although both events might count on the same cycle counted by STALL_BACKEND if there are both memory and processor-resource stall conditions active.

`0x8165`, **STALL_BACKEND_L1D, Backend stall cycles, level 1 data cache**

The counter counts each cycle counted by STALL_BACKEND_MEMBOUND when there is a demand data miss in the first level of data or unified cache.

The counter does not count the cycle if any of the following are true:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The STALL_BACKEND_L2D event is implemented and there is a demand data miss in the second level of data or unified cache, meaning the STALL_BACKEND_L2D event counts the cycle.
- There is a demand data miss in the last level of data or unified cache within the PE clock domain, meaning the STALL_BACKEND_MEM event counts the cycle.

`0x8166`, **STALL_BACKEND_L2D, Backend stall cycles, level 2 data cache**

The counter counts each cycle counted by STALL_BACKEND_MEMBOUND when there is a demand data miss in the second level of data or unified cache.

The counter does not count the cycle if there is a demand data miss in the last level of data or unified cache within the PE clock domain, meaning the STALL_BACKEND_MEM event counts the cycle.

`0x8167`, **STALL_BACKEND_TLB, Backend stall cycles, TLB**

The counter counts each cycle counted by STALL_BACKEND_MEMBOUND when there is a demand data miss in the data or unified TLB.

`0x8168`, **STALL_BACKEND_ST, Backend stall cycles, store**

The counter counts each cycle counted by STALL_BACKEND_MEMBOUND when the backend is stalled waiting for a store.

`0x816A`, **STALL_BACKEND_CPUBOUND, Backend stall cycles, processor bound**

The counter counts each cycle counted by STALL_BACKEND when the backend is stalled on a processor resource, not including memory.

The counter counts each stall that occurs when a backend processor resource is busy. This includes the stalls counted by STALL_BACKEND_RENAME, and any other IMPLEMENTATION DEFINED processor resource stalls.

It does not include stalls that are counted by STALL_BACKEND_MEMBOUND, although both events might count on the same cycle counted by STALL_BACKEND if there are both memory and processor-resource stall conditions active.

`0x816B`, **STALL_BACKEND_BUSY, Backend stall cycles, backend busy**

The counter counts each cycle counted by STALL_BACKEND when operations are available from the frontend but the backend is not able to accept an operation because an execution unit is busy.

For example a complex operation unit such as a divider is executing a previous operation and cannot accept a new operation.

`0x816C`, **STALL_BACKEND_ILOCK, Backend stall cycles, input dependency**

The counter counts each cycle counted by STALL_BACKEND when operations are available from the frontend but at least one is not ready to be sent to the backend because of an input dependency.

`0x816D`, **STALL_BACKEND_RENAME, Backend stall cycles, rename full**

The counter counts each cycle counted by STALL_BACKEND_CPUBOUND when operations are available from the frontend but at least one is not ready to be sent to the backend because no rename register is available.

If this event is implemented and counts such stalls then the STALL_FRONTEND_RENAME event counts as zero.

`0x816E`, **STALL_BACKEND_ATOMIC, Backend stall cycles, atomic operation**

The counter counts each cycle counted by STALL_BACKEND_MEMBOUND when the backend is processing an Atomic operation.

`0x8170`, **CAS_NEAR_FAIL, Atomic memory Operation speculatively executed, Compare and Swap fail**

The counter counts each Compare and Swap operation counted by CAS_NEAR_SPEC that does not update the location accessed.

`0x8171`, **CAS_NEAR_PASS, Atomic memory Operation speculatively executed, Compare and Swap pass**

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The counter counts each Compare and Swap operation counted by CAS_NEAR_SPEC that updates the location accessed.

`0x8172`, **CAS_NEAR_SPEC, Atomic memory Operation speculatively executed, Compare and Swap near**

The counter counts each Compare and Swap operation counted by CAS_SPEC that executes locally to the PE.

The definition of locally is IMPLEMENTATION DEFINED. Operations counted by CAS_NEAR_SPEC also generate one of the CAS_NEAR_PASS or CAS_NEAR_FAIL events.

`0x8173`, **CAS_FAR_SPEC, Atomic memory Operation speculatively executed, Compare and Swap far**

The counter counts each Compare and Swap operation counted by CAS_SPEC that does not execute locally to the PE.

The definition of locally is IMPLEMENTATION DEFINED. Operations counted by CAS_FAR_SPEC do not generate CAS_NEAR_PASS or CAS_NEAR_FAIL events.

`0x8174`, **CAS_SPEC, Atomic memory Operation speculatively executed, Compare and Swap**

The counter counts each Compare and Swap operation.

`0x8175`, **LSE_LD_SPEC, Atomic memory Operation speculatively executed, load**

The counter counts each Atomic memory operation counted by LSE_LDST_SPEC that returns a value to the PE, including Swap and Compare-and-Swap operations.

`0x8176`, **LSE_ST_SPEC, Atomic memory Operation speculatively executed, store**

The counter counts each Atomic memory operation counted by LSE_LDST_SPEC that does not return a value to the PE.

`0x8177`, **LSE_LDST_SPEC, Atomic memory Operation speculatively executed, load or store**

The counter counts each operation counted by LDST_SPEC that is an Atomic memory operation.

`0x8178`, **REMOTE_ACCESS_WR, Access to a remote device, write**

The counter counts each access counted by REMOTE_ACCESS that is a Memory-write operation.

`0x8186`, **UOP_RETIRED, Micro-operation architecturally executed**

The counter counts each micro-operation that would be executed in a Simple sequential execution of the program.

Unlike OP_RETIRED, this event is not linked to the definition of OP_SPEC, meaning it counts micro-operations that are created from other operations after those operations are counted by OP_SPEC.

`0x8188`, **DTLB_WALK_BLOCK, Data TLB block translation table walk**

The counter counts each translation table walk counted by DTLB_WALK where the result of the walk yields a Block.

That is, the translation table walk ends at a Block descriptor, at other than the last level of translation table.

If two stages of translation are used and Stage 1 and Stage 2 yield different granule sizes, the counter counts the translation table walk only if both are a Block.

The counter does not count the walk if the access generates a Translation fault.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x8189`, **ITLB_WALK_BLOCK, Instruction TLB block translation table walk**

The counter counts each translation table walk counted by ITLB_WALK where the result of the walk yields a Block.

That is, the translation table walk ends at a Page descriptor, at other than the last level of translation table.

If two stages of translation are used and Stage 1 and Stage 2 yield different granule sizes, the counter counts the translation table walk only if both are a Block.

The counter does not count the walk if the access generates a Translation fault.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x818A`, **DTLB_WALK_PAGE, Data TLB page translation table walk**

The counter counts each translation table walk counted by DTLB_WALK where the result of the walk yields a Page.

That is, the translation table walk ends at a Block descriptor, at the last level of translation table.

If two stages of translation are used and Stage 1 and Stage 2 yield different granule sizes, the counter counts the translation table walk only if either is a Page.

The counter does not count the walk if the access generates a Translation fault.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x818B`, **ITLB_WALK_PAGE, Instruction TLB page translation table walk**

The counter counts each translation table walk counted by ITLB_WALK where the result of the walk yields a Page.

That is, the translation table walk ends at a Page descriptor, at the last level of translation table.

If two stages of translation are used and Stage 1 and Stage 2 yield different granule sizes, the counter counts the translation table walk only if either is a Page.

The counter does not count the walk if the access generates a Translation fault.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x818D`, **BUS_REQ_RD, Bus request, read**

The counter counts each transaction counted by BUS_REQ that is a Memory-read operation.

`0x818E`, **BUS_REQ_WR, Bus request, write**

The counter counts each transaction counted by BUS_REQ that is a Memory-write operation.

`0x818F`, **BUS_REQ, Bus request**

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The counter counts each request generated by a Memory-read operation or Memory-write operation that accesses outside of the boundary of the PE and its closely-coupled caches.

Where this boundary lies with respect to any implemented caches is IMPLEMENTATION DEFINED. Where an implementation has multiple buses at this boundary, this event counts the sum of requests across all buses.

A bus request is the start of a bus transaction. The exact nature of a bus transaction is IMPLEMENTATION DEFINED, but for the purposes of event monitoring consists of a single access comprising one or more cycles, or beats, when the transaction occupies the bus. The BUS_REQ event therefore counts the number of transactions. The BUS_ACCESS event counts the occupancy of the transaction on the bus.

Bus transactions include refills of and write-backs from data, instruction, and unified caches. Whether bus transactions include operations that use the bus but do not explicitly transfer data is IMPLEMENTATION DEFINED. This must be the same for the BUS_REQ and BUS_ACCESS events.

An Unattributable bus transaction occurs when a requestor outside the PE makes a request that results in a bus transaction, for example, a coherency request.

The maximum increment in any given cycle is IMPLEMENTATION DEFINED.

If the bus is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the bus is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x8190`**, ISNP_HIT_RD, Snoop hit, demand instruction fetch**

The counter counts each snoop generated by the PE in response to a demand Instruction memory access that hits in a cache outside of the cache hierarchy of this PE.

The event is counted by the PE generating the snoop, not the PE being snooped.

`0x8191`**, ISNP_HIT_NEAR_RD, Snoop hit in near local cache, demand instruction fetch**

The counter counts each snoop generated by the PE in response to a demand Instruction memory access counted by ISNP_HIT_RD that hits in a cache outside of the cache hierarchy of this PE in the local PE cluster.

The event is counted by the PE generating the snoop, not the PE being snooped.

`0x8192`**, ISNP_HIT_FAR_RD, Snoop hit in far local cache, demand instruction fetch**

The counter counts each snoop generated by the PE in response to a demand Instruction memory access counted by ISNP_HIT_RD that hits in a cache outside the local PE cluster on the same device.

The event is counted by the PE generating the snoop, not the PE being snooped.

`0x8193`**, ISNP_HIT_REMOTE_RD, Snoop hit in remote cache, demand instruction fetch**

The counter counts each snoop generated by the PE in response to a demand Instruction memory access counted by ISNP_HIT_RD that hits in a cache on a remote device.

The event is counted by the PE generating the snoop, not the PE being snooped.

`0x8194`**, DSNP_HIT_RD, Snoop hit, demand data read**

The counter counts each snoop generated by the PE in response to a demand Memory-read operation counted by DSNP_HIT_RW that hits in a cache outside of the cache hierarchy of this PE.

The event is counted by the PE generating the snoop, not the PE being snooped.

`0x8195`**, DSNP_HIT_NEAR_RD, Snoop hit in near local cache, demand data read**

The counter counts each snoop generated by the PE in response to a demand Memory-read operation counted by DSNP_HIT_RD that hits in a cache outside of the cache hierarchy of this PE in the local PE cluster.

The event is counted by the PE generating the snoop, not the PE being snooped.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x8196`, **DSNP_HIT_FAR_RD, Snoop hit in far local cache, demand data read**

The counter counts each snoop generated by the PE in response to a demand Memory-read operation counted by DSNP_HIT_RD that hits in a cache outside the local PE cluster on the same device.

The event is counted by the PE generating the snoop, not the PE being snooped.

`0x8197`, **DSNP_HIT_REMOTE_RD, Snoop hit in remote cache, demand data read**

The counter counts each snoop generated by the PE in response to a demand Memory-read operation counted by DSNP_HIT_RD that hits in a cache on a remote device.

The event is counted by the PE generating the snoop, not the PE being snooped.

`0x8198`, **DSNP_HIT_WR, Snoop hit, demand data write**

The counter counts each snoop generated by the PE in response to a demand Memory-write operation counted by DSNP_HIT_RW that hits in a cache outside of the cache hierarchy of this PE.

The event is counted by the PE generating the snoop, not the PE being snooped.

`0x8199`, **DSNP_HIT_NEAR_WR, Snoop hit in near local cache, demand data write**

The counter counts each snoop generated by the PE in response to a demand Memory-write operation counted by DSNP_HIT_WR that hits in a cache outside of the cache hierarchy of this PE in the local PE cluster.

The event is counted by the PE generating the snoop, not the PE being snooped.

`0x819A`, **DSNP_HIT_FAR_WR, Snoop hit in far local cache, demand data write**

The counter counts each snoop generated by the PE in response to a demand Memory-write operation counted by DSNP_HIT_WR that hits in a cache outside the local PE cluster on the same device.

The event is counted by the PE generating the snoop, not the PE being snooped.

`0x819B`, **DSNP_HIT_REMOTE_WR, Snoop hit in remote cache, demand data write**

The counter counts each snoop generated by the PE in response to a demand Memory-write operation counted by DSNP_HIT_WR that hits in a cache on a remote device.

The event is counted by the PE generating the snoop, not the PE being snooped.

`0x819C`, **DSNP_HIT_RW, Snoop hit, demand data access**

The counter counts each snoop generated by the PE in response to a demand Memory-read operation or demand Memory-write operation that hits in a cache outside of the cache hierarchy of this PE.

The event is counted by the PE generating the snoop, not the PE being snooped.

`0x819D`, **DSNP_HIT_NEAR_RW, Snoop hit in near local cache, demand data access**

The counter counts each snoop generated by the PE in response to a demand Memory-read operation or demand Memory-write operation counted by DSNP_HIT_RW that hits in a cache outside of the cache hierarchy of this PE in the local PE cluster.

The event is counted by the PE generating the snoop, not the PE being snooped.

`0x819E`, **DSNP_HIT_FAR_RW, Snoop hit in far local cache, demand data access**

The counter counts each snoop generated by the PE in response to a demand Memory-read operation or demand Memory-write operation counted by DSNP_HIT_RW that hits in a cache outside the local PE cluster on the same device.

The event is counted by the PE generating the snoop, not the PE being snooped.

`0x819F`, **DSNP_HIT_REMOTE_RW, Snoop hit in remote cache, demand data access**

The counter counts each snoop generated by the PE in response to a demand Memory-read operation or demand Memory-write operation counted by DSNP_HIT_RW that hits in a cache on a remote device.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The event is counted by the PE generating the snoop, not the PE being snooped.

`0x81A0`, **DSNP_HIT_PRFM, Snoop hit, software data prefetch**

The counter counts each snoop generated by the PE in response to a data prefetch instruction counted by DSNP_HIT_PRF that hits in a cache outside of the cache hierarchy of this PE.

The event is counted by the PE generating the snoop, not the PE being snooped.

`0x81A1`, **DSNP_HIT_NEAR_PRFM, Snoop hit in near local cache, software data prefetch**

The counter counts each snoop generated by the PE in response to a data prefetch instruction counted by DSNP_HIT_PRFM that hits in a cache outside of the cache hierarchy of this PE in the local PE cluster.

The event is counted by the PE generating the snoop, not the PE being snooped.

`0x81A2`, **DSNP_HIT_FAR_PRFM, Snoop hit in far local cache, software data prefetch**

The counter counts each snoop generated by the PE in response to a data prefetch instruction counted by DSNP_HIT_PRFM that hits in a cache outside the local PE cluster on the same device.

The event is counted by the PE generating the snoop, not the PE being snooped.

`0x81A3`, **DSNP_HIT_REMOTE_PRFM, Snoop hit in remote cache, software data prefetch**

The counter counts each snoop generated by the PE in response to a data prefetch instruction counted by DSNP_HIT_PRFM that hits in a cache on a remote device.

The event is counted by the PE generating the snoop, not the PE being snooped.

`0x81A4`, **DSNP_HIT_HWPRF, Snoop hit, hardware data prefetch**

The counter counts each snoop generated by a data hardware prefetch counted by DSNP_HIT_PRF that hits in a cache outside of the cache hierarchy of this PE.

The event is counted by the PE generating the snoop, not the PE being snooped.

`0x81A5`, **DSNP_HIT_NEAR_HWPRF, Snoop hit in near local cache, hardware data prefetch**

The counter counts each snoop generated by a data hardware prefetch counted by DSNP_HIT_HWPRF that hits in a cache outside of the cache hierarchy of this PE in the local PE cluster.

The event is counted by the PE generating the snoop, not the PE being snooped.

`0x81A6`, **DSNP_HIT_FAR_HWPRF, Snoop hit in far local cache, hardware data prefetch**

The counter counts each snoop generated by a data hardware prefetch counted by DSNP_HIT_HWPRF that hits in a cache outside the local PE cluster on the same device.

The event is counted by the PE generating the snoop, not the PE being snooped.

`0x81A7`, **DSNP_HIT_REMOTE_HWPRF, Snoop hit in remote cache, hardware data prefetch**

The counter counts each snoop generated by a data hardware prefetch counted by DSNP_HIT_HWPRF that hits in a cache on a remote device.

The event is counted by the PE generating the snoop, not the PE being snooped.

`0x81A8`, **ISNP_HIT_PRFM, Snoop hit, software instruction prefetch**

The counter counts each snoop generated by the PE in response to an instruction prefetch instruction counted by ISNP_HIT_PRF that hits in a cache outside of the cache hierarchy of this PE.

The event is counted by the PE generating the snoop, not the PE being snooped.

`0x81A9`, **ISNP_HIT_NEAR_PRFM, Snoop hit in near local cache, software instruction prefetch**

The counter counts each snoop generated by the PE in response to an instruction prefetch instruction counted by ISNP_HIT_PRFM that hits in a cache outside of the cache hierarchy of this PE in the local PE cluster.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The event is counted by the PE generating the snoop, not the PE being snooped.

`0x81AA`, **ISNP_HIT_FAR_PRFM, Snoop hit in far local cache, software instruction prefetch**

The counter counts each snoop generated by the PE in response to an instruction prefetch instruction counted by ISNP_HIT_PRFM that hits in a cache outside the local PE cluster on the same device.

The event is counted by the PE generating the snoop, not the PE being snooped.

`0x81AB`, **ISNP_HIT_REMOTE_PRFM, Snoop hit in remote cache, software instruction prefetch**

The counter counts each snoop generated by the PE in response to an instruction prefetch instruction counted by ISNP_HIT_PRFM that hits in a cache on a remote device.

The event is counted by the PE generating the snoop, not the PE being snooped.

`0x81AC`, **ISNP_HIT_HWPRF, Snoop hit, hardware instruction prefetch**

The counter counts each snoop generated by an instruction hardware prefetch counted by ISNP_HIT_PRF that hits in a cache outside of the cache hierarchy of this PE.

The event is counted by the PE generating the snoop, not the PE being snooped.

`0x81AD`, **ISNP_HIT_NEAR_HWPRF, Snoop hit in near local cache, hardware instruction prefetch**

The counter counts each snoop generated by an instruction hardware prefetch counted by ISNP_HIT_HWPRF that hits in a cache outside of the cache hierarchy of this PE in the local PE cluster.

The event is counted by the PE generating the snoop, not the PE being snooped.

`0x81AE`, **ISNP_HIT_FAR_HWPRF, Snoop hit in far local cache, hardware instruction prefetch**

The counter counts each snoop generated by an instruction hardware prefetch counted by ISNP_HIT_HWPRF that hits in a cache outside the local PE cluster on the same device.

The event is counted by the PE generating the snoop, not the PE being snooped.

`0x81AF`, **ISNP_HIT_REMOTE_HWPRF, Snoop hit in remote cache, hardware instruction prefetch**

The counter counts each snoop generated by an instruction hardware prefetch counted by ISNP_HIT_HWPRF that hits in a cache on a remote device.

The event is counted by the PE generating the snoop, not the PE being snooped.

`0x81B0`, **ISNP_HIT, Snoop hit, instruction**

The counter counts each snoop that hits in a cache outside of the cache hierarchy of this PE.

The event is counted by the PE generating the snoop, not the PE being snooped.

`0x81B1`, **ISNP_HIT_NEAR, Snoop hit in near local cache, instruction access**

The counter counts each snoop counted by ISNP_HIT that hits in a cache outside of the cache hierarchy of this PE in the local PE cluster.

The event is counted by the PE generating the snoop, not the PE being snooped.

`0x81B2`, **ISNP_HIT_FAR, Snoop hit in far local cache, instruction access**

The counter counts each snoop counted by ISNP_HIT that hits in a cache outside the local PE cluster on the same device.

The event is counted by the PE generating the snoop, not the PE being snooped.

`0x81B3`, **ISNP_HIT_REMOTE, Snoop hit in remote cache, instruction access**

The counter counts each snoop counted by ISNP_HIT that hits in a cache on a remote device.

The event is counted by the PE generating the snoop, not the PE being snooped.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x81B4`, **DSNP_HIT, Snoop hit, data**

The counter counts each snoop that hits in a cache outside of the cache hierarchy of this PE.

The event is counted by the PE generating the snoop, not the PE being snooped.

`0x81B5`, **DSNP_HIT_NEAR, Snoop hit in near local cache, data access**

The counter counts each snoop counted by DSNP_HIT that hits in a cache outside of the cache hierarchy of this PE in the local PE cluster.

The event is counted by the PE generating the snoop, not the PE being snooped.

`0x81B6`, **DSNP_HIT_FAR, Snoop hit in far local cache, data access**

The counter counts each snoop counted by DSNP_HIT that hits in a cache outside the local PE cluster on the same device.

The event is counted by the PE generating the snoop, not the PE being snooped.

`0x81B7`, **DSNP_HIT_REMOTE, Snoop hit in remote cache, data access**

The counter counts each snoop counted by DSNP_HIT that hits in a cache on a remote device.

The event is counted by the PE generating the snoop, not the PE being snooped.

`0x81B8`, **L1I_CACHE_REFILL_HWPRF, Level 1 instruction cache refill, hardware prefetch**

The counter counts each hardware prefetch counted by L1I_CACHE_HWPRF that causes a refill of the Level 1 instruction or unified cache from outside of the Level 1 instruction or unified cache.

`0x81B9`, **L2I_CACHE_REFILL_HWPRF, Level 2 instruction cache refill, hardware prefetch**

The counter counts each hardware prefetch counted by L2I_CACHE_HWPRF that causes a refill of the Level 2 instruction or unified cache, or any Level 1 data, instruction, or unified cache of this PE, from outside of those caches.

`0x81BC`, **L1D_CACHE_REFILL_HWPRF, Level 1 data cache refill, hardware prefetch**

The counter counts each hardware prefetch counted by L1D_CACHE_HWPRF that causes a refill of the Level 1 data or unified cache from outside of the Level 1 data or unified cache.

`0x81BD`, **L2D_CACHE_REFILL_HWPRF, Level 2 data cache refill, hardware prefetch**

The counter counts each hardware prefetch counted by L2D_CACHE_HWPRF that causes a refill of the Level 2 data or unified cache, or any Level 1 data, instruction, or unified cache of this PE, from outside of those caches.

`0x81BE`, **L3D_CACHE_REFILL_HWPRF, Level 3 data cache refill, hardware prefetch**

The counter counts each hardware prefetch counted by L3D_CACHE_HWPRF that causes a refill of the Level 3 data or unified cache, or any Level 1 or Level 2 data, instruction, or unified cache of this PE, from outside of those caches.

`0x81BF`, **LL_CACHE_REFILL_HWPRF, Last level cache refill, hardware prefetch**

The counter counts each hardware prefetch counted by LL_CACHE_HWPRF that causes a refill of the Last level cache, or any other data, instruction, or unified cache of this PE, from outside of those caches.

`0x81C0`, **L1I_CACHE_HIT_RD, Level 1 instruction cache demand fetch hit**

The counter counts each demand fetch counted by L1I_CACHE_RD that hits in the Level 1 instruction or unified cache.

`0x81C1`, **L2I_CACHE_HIT_RD, Level 2 instruction cache demand fetch hit**

The counter counts each demand fetch counted by L2I_CACHE_RD that hits in the Level 2 instruction or unified cache.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x81C4`, **L1D_CACHE_HIT_RD, Level 1 data cache demand access hit, read**

The counter counts each demand read counted by L1D_CACHE_RD that hits in the Level 1 data or unified cache.

`0x81C5`, **L2D_CACHE_HIT_RD, Level 2 data cache demand access hit, read**

The counter counts each demand read counted by L2D_CACHE_RD that hits in the Level 2 data or unified cache.

`0x81C6`, **L3D_CACHE_HIT_RD, Level 3 data cache demand access hit, read**

The counter counts each demand read counted by L3D_CACHE_RD that hits in the Level 3 data or unified cache.

`0x81C7`, **LL_CACHE_HIT_RD, Last level cache demand access hit, read**

The counter counts each demand read counted by LL_CACHE_RD that hits in the Last level cache.

`0x81C8`, **L1D_CACHE_HIT_WR, Level 1 data cache demand access hit, write**

The counter counts each demand write counted by L1D_CACHE_WR that hits in the Level 1 data or unified cache.

`0x81C9`, **L2D_CACHE_HIT_WR, Level 2 data cache demand access hit, write**

The counter counts each demand write counted by L2D_CACHE_WR that hits in the Level 2 data or unified cache.

`0x81CA`, **L3D_CACHE_HIT_WR, Level 3 data cache demand access hit, write**

The counter counts each demand write counted by L3D_CACHE_WR that hits in the Level 3 data or unified cache.

`0x81CB`, **LL_CACHE_HIT_WR, Last level cache demand access hit, write**

The counter counts each demand write counted by LL_CACHE_WR that hits in the Last level cache.

`0x81CC`, **L1D_CACHE_HIT_RW, Level 1 data cache demand access hit**

The counter counts each demand access counted by L1D_CACHE_RW that hits in the Level 1 data or unified cache.

`0x81CD`, **L2D_CACHE_HIT_RW, Level 2 data cache demand access hit**

The counter counts each demand access counted by L2D_CACHE_RW that hits in the Level 2 data or unified cache.

`0x81CE`, **L3D_CACHE_HIT_RW, Level 3 data cache demand access hit**

The counter counts each demand access counted by L3D_CACHE_RW that hits in the Level 3 data or unified cache.

`0x81CF`, **LL_CACHE_HIT_RW, Last level cache demand access hit**

The counter counts each demand access counted by LL_CACHE_RW that hits in the Last level cache.

`0x81D0`, **L1I_CACHE_HIT_RD_FPRFM, Level 1 instruction cache demand fetch first hit, fetched by software prefetch**

The counter counts each demand fetch first hit counted by L1I_CACHE_HIT_RD_FPRF where the cache line was fetched in response to a prefetch instruction.

That is, the L1I_CACHE_REFILL_PRFM event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x81D1`, **L2I_CACHE_HIT_RD_FPRFM, Level 2 instruction cache demand fetch first hit, fetched by software prefetch**

The counter counts each demand fetch first hit counted by L2I_CACHE_HIT_RD_FPRF where the cache line was fetched in response to a prefetch instruction.

That is, the L2I_CACHE_REFILL_PRFM event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

`0x81D4`, **L1D_CACHE_HIT_RD_FPRFM, Level 1 data cache demand access first hit, read, fetched by software prefetch**

The counter counts each first hit counted by L1D_CACHE_HIT_RW_FPRFM that is due to a demand Memory-read operation.

`0x81D5`, **L2D_CACHE_HIT_RD_FPRFM, Level 2 data cache demand access first hit, read, fetched by software prefetch**

The counter counts each first hit counted by L2D_CACHE_HIT_RW_FPRFM that is due to a demand Memory-read operation.

`0x81D6`, **L3D_CACHE_HIT_RD_FPRFM, Level 3 data cache demand access first hit, read, fetched by software prefetch**

The counter counts each first hit counted by L3D_CACHE_HIT_RW_FPRFM that is due to a demand Memory-read operation.

`0x81D7`, **LL_CACHE_HIT_RD_FPRFM, Last level cache demand access first hit, read, fetched by software prefetch**

The counter counts each first hit counted by LL_CACHE_HIT_RW_FPRFM that is due to a demand Memory-read operation.

`0x81D8`, **L1D_CACHE_HIT_WR_FPRFM, Level 1 data cache demand access first hit, write, fetched by software prefetch**

The counter counts each first hit counted by L1D_CACHE_HIT_RW_FPRFM that is due to a demand Memory-write operation.

`0x81D9`, **L2D_CACHE_HIT_WR_FPRFM, Level 2 data cache demand access first hit, write, fetched by software prefetch**

The counter counts each first hit counted by L2D_CACHE_HIT_RW_FPRFM that is due to a demand Memory-write operation.

`0x81DA`, **L3D_CACHE_HIT_WR_FPRFM, Level 3 data cache demand access first hit, write, fetched by software prefetch**

The counter counts each first hit counted by L3D_CACHE_HIT_RW_FPRFM that is due to a demand Memory-write operation.

`0x81DB`, **LL_CACHE_HIT_WR_FPRFM, Last level cache demand access first hit, write, fetched by software prefetch**

The counter counts each first hit counted by LL_CACHE_HIT_RW_FPRFM that is due to a demand Memory-write operation.

`0x81DC`, **L1D_CACHE_HIT_RW_FPRFM, Level 1 data cache demand access first hit, fetched by software prefetch**

The counter counts each demand access first hit counted by L1D_CACHE_HIT_RW_FPRF where the cache line was fetched in response to a prefetch instruction.

That is, the L1D_CACHE_REFILL_PRFM event was generated when the cache line was fetched into the cache.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

`0x81DD`, **L2D_CACHE_HIT_RW_FPRFM, Level 2 data cache demand access first hit, fetched by software prefetch**

The counter counts each demand access first hit counted by L2D_CACHE_HIT_RW_FPRF where the cache line was fetched in response to a prefetch instruction.

That is, the L2D_CACHE_REFILL_PRFM event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

`0x81DE`, **L3D_CACHE_HIT_RW_FPRFM, Level 3 data cache demand access first hit, fetched by software prefetch**

The counter counts each demand access first hit counted by L3D_CACHE_HIT_RW_FPRF where the cache line was fetched in response to a prefetch instruction.

That is, the L3D_CACHE_REFILL_PRFM event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

`0x81DF`, **LL_CACHE_HIT_RW_FPRFM, Last level cache demand access first hit, fetched by software prefetch**

The counter counts each demand access first hit counted by LL_CACHE_HIT_RW_FPRF where the cache line was fetched in response to a prefetch instruction.

That is, the LL_CACHE_REFILL_PRFM event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

`0x81E0`, **L1I_CACHE_HIT_RD_FHWPRF, Level 1 instruction cache demand fetch first hit, fetched by hardware prefetcher**

The counter counts each demand fetch first hit counted by L1I_CACHE_HIT_RD_FPRF where the cache line was fetched by a hardware prefetcher.

That is, the L1I_CACHE_REFILL_HWPRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

`0x81E1`, **L2I_CACHE_HIT_RD_FHWPRF, Level 2 instruction cache demand fetch first hit, fetched by hardware prefetcher**

The counter counts each demand fetch first hit counted by L2I_CACHE_HIT_RD_FPRF where the cache line was fetched by a hardware prefetcher.

That is, the L2I_CACHE_REFILL_HWPRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

`0x81E4`, **L1D_CACHE_HIT_RD_FHWPRF, Level 1 data cache demand access first hit, read, fetched by hardware prefetcher**

The counter counts each first hit counted by L1D_CACHE_HIT_RW_FHWPRF that is due to a demand Memory-read operation.

`0x81E5`, **L2D_CACHE_HIT_RD_FHWPRF, Level 2 data cache demand access first hit, read, fetched by hardware prefetcher**

The counter counts each first hit counted by L2D_CACHE_HIT_RW_FHWPRF that is due to a demand Memory-read operation.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x81E6`, **L3D_CACHE_HIT_RD_FHWPRF, Level 3 data cache demand access first hit, read, fetched by hardware prefetcher**

The counter counts each first hit counted by L3D_CACHE_HIT_RW_FHWPRF that is due to a demand Memory-read operation.

`0x81E7`, **LL_CACHE_HIT_RD_FHWPRF, Last level cache demand access first hit, read, fetched by hardware prefetcher**

The counter counts each first hit counted by LL_CACHE_HIT_RW_FHWPRF that is due to a demand Memory-read operation.

`0x81E8`, **L1D_CACHE_HIT_WR_FHWPRF, Level 1 data cache demand access first hit, write, fetched by hardware prefetcher**

The counter counts each first hit counted by L1D_CACHE_HIT_RW_FHWPRF that is due to a demand Memory-write operation.

`0x81E9`, **L2D_CACHE_HIT_WR_FHWPRF, Level 2 data cache demand access first hit, write, fetched by hardware prefetcher**

The counter counts each first hit counted by L2D_CACHE_HIT_RW_FHWPRF that is due to a demand Memory-write operation.

`0x81EA`, **L3D_CACHE_HIT_WR_FHWPRF, Level 3 data cache demand access first hit, write, fetched by hardware prefetcher**

The counter counts each first hit counted by L3D_CACHE_HIT_RW_FHWPRF that is due to a demand Memory-write operation.

`0x81EB`, **LL_CACHE_HIT_WR_FHWPRF, Last level cache demand access first hit, write, fetched by hardware prefetcher**

The counter counts each first hit counted by LL_CACHE_HIT_RW_FHWPRF that is due to a demand Memory-write operation.

`0x81EC`, **L1D_CACHE_HIT_RW_FHWPRF, Level 1 data cache demand access first hit, fetched by hardware prefetcher**

The counter counts each demand access first hit counted by L1D_CACHE_HIT_RW_FPRF where the cache line was fetched by a hardware prefetcher.

That is, the L1D_CACHE_REFILL_HWPRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

`0x81ED`, **L2D_CACHE_HIT_RW_FHWPRF, Level 2 data cache demand access first hit, fetched by hardware prefetcher**

The counter counts each demand access first hit counted by L2D_CACHE_HIT_RW_FPRF where the cache line was fetched by a hardware prefetcher.

That is, the L2D_CACHE_REFILL_HWPRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

`0x81EE`, **L3D_CACHE_HIT_RW_FHWPRF, Level 3 data cache demand access first hit, fetched by hardware prefetcher**

The counter counts each demand access first hit counted by L3D_CACHE_HIT_RW_FPRF where the cache line was fetched by a hardware prefetcher.

That is, the L3D_CACHE_REFILL_HWPRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x81EF`, **LL_CACHE_HIT_RW_FHWPRF, Last level cache demand access first hit, fetched by hardware prefetcher**

The counter counts each demand access first hit counted by LL_CACHE_HIT_RW_FPRF where the cache line was fetched by a hardware prefetcher.

That is, the LL_CACHE_REFILL_HWPRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

`0x81F0`, **L1I_CACHE_HIT_RD_FPRF, Level 1 instruction cache demand fetch first hit, fetched by prefetch**

The counter counts each demand fetch first hit counted by L1I_CACHE_HIT_RD where the cache line was fetched in response to a prefetch instruction or by a hardware prefetcher.

That is, the L1I_CACHE_REFILL_PRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

`0x81F1`, **L2I_CACHE_HIT_RD_FPRF, Level 2 instruction cache demand fetch first hit, fetched by prefetch**

The counter counts each demand fetch first hit counted by L2I_CACHE_HIT_RD where the cache line was fetched in response to a prefetch instruction or by a hardware prefetcher.

That is, the L2I_CACHE_REFILL_PRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

`0x81F4`, **L1D_CACHE_HIT_RD_FPRF, Level 1 data cache demand access first hit, read, fetched by prefetch**

The counter counts each first hit counted by L1D_CACHE_HIT_RW_FPRF that is due to a demand Memory-read operation.

`0x81F5`, **L2D_CACHE_HIT_RD_FPRF, Level 2 data cache demand access first hit, read, fetched by prefetch**

The counter counts each first hit counted by L2D_CACHE_HIT_RW_FPRF that is due to a demand Memory-read operation.

`0x81F6`, **L3D_CACHE_HIT_RD_FPRF, Level 3 data cache demand access first hit, read, fetched by prefetch**

The counter counts each first hit counted by L3D_CACHE_HIT_RW_FPRF that is due to a demand Memory-read operation.

`0x81F7`, **LL_CACHE_HIT_RD_FPRF, Last level cache demand access first hit, read, fetched by prefetch**

The counter counts each first hit counted by LL_CACHE_HIT_RW_FPRF that is due to a demand Memory-read operation.

`0x81F8`, **L1D_CACHE_HIT_WR_FPRF, Level 1 data cache demand access first hit, write, fetched by prefetch**

The counter counts each first hit counted by L1D_CACHE_HIT_RW_FPRF that is due to a demand Memory-write operation.

`0x81F9`, **L2D_CACHE_HIT_WR_FPRF, Level 2 data cache demand access first hit, write, fetched by prefetch**

The counter counts each first hit counted by L2D_CACHE_HIT_RW_FPRF that is due to a demand Memory-write operation.

`0x81FA`, **L3D_CACHE_HIT_WR_FPRF, Level 3 data cache demand access first hit, write, fetched by prefetch**

The counter counts each first hit counted by L3D_CACHE_HIT_RW_FPRF that is due to a demand Memory-write operation.

`0x81FB`, **LL_CACHE_HIT_WR_FPRF, Last level cache demand access first hit, write, fetched by prefetch**

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The counter counts each first hit counted by LL_CACHE_HIT_RW_FPRF that is due to a demand Memory-write operation.

`0x81FC`, **L1D_CACHE_HIT_RW_FPRF, Level 1 data cache demand access first hit, fetched by prefetch**

The counter counts each demand access first hit counted by L1D_CACHE_HIT_RW where the cache line was fetched in response to a prefetch instruction or by a hardware prefetcher.

That is, the L1D_CACHE_REFILL_PRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

`0x81FD`, **L2D_CACHE_HIT_RW_FPRF, Level 2 data cache demand access first hit, fetched by prefetch**

The counter counts each demand access first hit counted by L2D_CACHE_HIT_RW where the cache line was fetched in response to a prefetch instruction or by a hardware prefetcher.

That is, the L2D_CACHE_REFILL_PRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

`0x81FE`, **L3D_CACHE_HIT_RW_FPRF, Level 3 data cache demand access first hit, fetched by prefetch**

The counter counts each demand access first hit counted by L3D_CACHE_HIT_RW where the cache line was fetched in response to a prefetch instruction or by a hardware prefetcher.

That is, the L3D_CACHE_REFILL_PRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

`0x81FF`, **LL_CACHE_HIT_RW_FPRF, Last level cache demand access first hit, fetched by prefetch**

The counter counts each demand access first hit counted by LL_CACHE_HIT_RW where the cache line was fetched in response to a prefetch instruction or by a hardware prefetcher.

That is, the LL_CACHE_REFILL_PRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

`0x8200`, **L1I_CACHE_HIT, Level 1 instruction cache hit**

The counter counts each access counted by L1I_CACHE that hits in the Level 1 instruction or unified cache.

`0x8201`, **L2I_CACHE_HIT, Level 2 instruction cache hit**

The counter counts each access counted by L2I_CACHE that hits in the Level 2 instruction or unified cache.

`0x8204`, **L1D_CACHE_HIT, Level 1 data cache hit**

The counter counts each access counted by L1D_CACHE that hits in the Level 1 data or unified cache.

`0x8205`, **L2D_CACHE_HIT, Level 2 data cache hit**

The counter counts each access counted by L2D_CACHE that hits in the Level 2 data or unified cache.

`0x8206`, **L3D_CACHE_HIT, Level 3 data cache hit**

The counter counts each access counted by L3D_CACHE that hits in the Level 3 data or unified cache.

`0x8207`, **LL_CACHE_HIT, Last level cache hit**

The counter counts each access counted by LL_CACHE that hits in the Last level cache.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x8208`, **L1I_CACHE_HIT_PRFM, Level 1 instruction cache software prefetch hit**

The counter counts each software prefetch counted by L1I_CACHE_PRFM that hits in the Level 1 instruction or unified cache.

`0x8209`, **L2I_CACHE_HIT_PRFM, Level 2 instruction cache software prefetch hit**

The counter counts each software prefetch counted by L2I_CACHE_PRFM that hits in the Level 2 instruction or unified cache.

`0x820C`, **L1D_CACHE_HIT_PRFM, Level 1 data cache software prefetch hit**

The counter counts each software prefetch counted by L1D_CACHE_PRFM that hits in the Level 1 data or unified cache.

`0x820D`, **L2D_CACHE_HIT_PRFM, Level 2 data cache software prefetch hit**

The counter counts each software prefetch counted by L2D_CACHE_PRFM that hits in the Level 2 data or unified cache.

`0x820E`, **L3D_CACHE_HIT_PRFM, Level 3 data cache software prefetch hit**

The counter counts each software prefetch counted by L3D_CACHE_PRFM that hits in the Level 3 data or unified cache.

`0x820F`, **LL_CACHE_HIT_PRFM, Last level cache software prefetch hit**

The counter counts each software prefetch counted by LL_CACHE_PRFM that hits in the Last level cache.

`0x8214`, **L1D_CACHE_HITM_RD, Level 1 data cache demand access hit modified, read**

The counter counts each cache access counted by L1D_CACHE_HIT_RD that accesses a previously modified cache location.

`0x8215`, **L2D_CACHE_HITM_RD, Level 2 data cache demand access hit modified, read**

The counter counts each cache access counted by L2D_CACHE_HIT_RD that accesses a previously modified cache location.

`0x8216`, **L3D_CACHE_HITM_RD, Level 3 data cache demand access hit modified, read**

The counter counts each cache access counted by L3D_CACHE_HIT_RD that accesses a previously modified cache location.

`0x8217`, **LL_CACHE_HITM_RD, Last level cache demand access hit modified, read**

The counter counts each cache access counted by LL_CACHE_HIT_RD that accesses a previously modified cache location.

`0x8218`, **L1D_CACHE_HITM_WR, Level 1 data cache demand access hit modified, write**

The counter counts each cache access counted by L1D_CACHE_HIT_WR that accesses a previously modified cache location.

`0x8219`, **L2D_CACHE_HITM_WR, Level 2 data cache demand access hit modified, write**

The counter counts each cache access counted by L2D_CACHE_HIT_WR that accesses a previously modified cache location.

`0x821A`, **L3D_CACHE_HITM_WR, Level 3 data cache demand access hit modified, write**

The counter counts each cache access counted by L3D_CACHE_HIT_WR that accesses a previously modified cache location.

`0x821B`, **LL_CACHE_HITM_WR, Last level cache demand access hit modified, write**

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The counter counts each cache access counted by LL_CACHE_HIT_WR that accesses a previously modified cache location.

`0x821C`, **L1D_CACHE_HITM_RW, Level 1 data cache demand access hit modified**

The counter counts each cache access counted by L1D_CACHE_HIT_RW that accesses a previously modified cache location.

`0x821D`, **L2D_CACHE_HITM_RW, Level 2 data cache demand access hit modified**

The counter counts each cache access counted by L2D_CACHE_HIT_RW that accesses a previously modified cache location.

`0x821E`, **L3D_CACHE_HITM_RW, Level 3 data cache demand access hit modified**

The counter counts each cache access counted by L3D_CACHE_HIT_RW that accesses a previously modified cache location.

`0x821F`, **LL_CACHE_HITM_RW, Last level cache demand access hit modified**

The counter counts each cache access counted by LL_CACHE_HIT_RW that accesses a previously modified cache location.

`0x8224`, **DSNP_HITM_RD, Snoop hit, demand data read, modified**

The counter counts each snoop generated by the PE in response to a demand Memory-read operation counted by DSNP_HIT_RD that accesses a previously modified cache location.

`0x8225`, **DSNP_HITM_NEAR_RD, Snoop hit in near local cache, demand data read, modified**

The counter counts each snoop generated by the PE in response to a demand Memory-read operation counted by DSNP_HIT_NEAR_RD that accesses a previously modified cache location.

`0x8226`, **DSNP_HITM_FAR_RD, Snoop hit in far local cache, demand data read, modified**

The counter counts each snoop generated by the PE in response to a demand Memory-read operation counted by DSNP_HIT_FAR_RD that accesses a previously modified cache location.

`0x8227`, **DSNP_HITM_REMOTE_RD, Snoop hit in remote cache, demand data read, modified**

The counter counts each snoop generated by the PE in response to a demand Memory-read operation counted by DSNP_HIT_REMOTE_RD that accesses a previously modified cache location.

`0x8228`, **DSNP_HITM_WR, Snoop hit, demand data write, modified**

The counter counts each snoop generated by the PE in response to a demand Memory-write operation counted by DSNP_HIT_WR that accesses a previously modified cache location.

`0x8229`, **DSNP_HITM_NEAR_WR, Snoop hit in near local cache, demand data write, modified**

The counter counts each snoop generated by the PE in response to a demand Memory-write operation counted by DSNP_HIT_NEAR_WR that accesses a previously modified cache location.

`0x822A`, **DSNP_HITM_FAR_WR, Snoop hit in far local cache, demand data write, modified**

The counter counts each snoop generated by the PE in response to a demand Memory-write operation counted by DSNP_HIT_FAR_WR that accesses a previously modified cache location.

`0x822B`, **DSNP_HITM_REMOTE_WR, Snoop hit in remote cache, demand data write, modified**

The counter counts each snoop generated by the PE in response to a demand Memory-write operation counted by DSNP_HIT_REMOTE_WR that accesses a previously modified cache location.

`0x822C`, **DSNP_HITM_RW, Snoop hit, demand data access, modified**

The counter counts each snoop generated by the PE in response to a demand Memory-read operation or demand Memory-write operation counted by DSNP_HIT_RW that accesses a previously modified cache location.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x822D`, **DSNP_HITM_NEAR_RW, Snoop hit in near local cache, demand data access, modified**

The counter counts each snoop generated by the PE in response to a demand Memory-read operation or demand Memory-write operation counted by DSNP_HIT_NEAR_RW that accesses a previously modified cache location.

`0x822E`, **DSNP_HITM_FAR_RW, Snoop hit in far local cache, demand data access, modified**

The counter counts each snoop generated by the PE in response to a demand Memory-read operation or demand Memory-write operation counted by DSNP_HIT_FAR_RW that accesses a previously modified cache location.

`0x822F`, **DSNP_HITM_REMOTE_RW, Snoop hit in remote cache, demand data access, modified**

The counter counts each snoop generated by the PE in response to a demand Memory-read operation or demand Memory-write operation counted by DSNP_HIT_REMOTE_RW that accesses a previously modified cache location.

`0x8230`, **LOCAL_MEM, Access to memory attached to this device**

The counter counts each Memory-read operation or Memory-write operation access to external memory attached to this device.

In a multi-socket system this means attached to the socket that contains the PE. For more information, see REMOTE_ACCESS.

In systems where there multiple types of memory attached to this device with different performance characteristics, it is IMPLEMENTATION DEFINED whether accesses to all external memory types are counted by this event, or are classified as remote accesses and counted by REMOTE_MEM.

For example, in a system with an expansion memory connected to the device that has significantly higher latency than the main system memory, accesses to the main system memory might be counted by this event and accesses to the expansion memory by REMOTE_MEM.

`0x8231`, **LOCAL_MEM_RD, Access to memory attached to this device, demand access, read**

The counter counts each access counted by LOCAL_MEM_RW that is a demand Memory-read operation.

`0x8232`, **LOCAL_MEM_WR, Access to memory attached to this device, demand access, write**

The counter counts each access counted by LOCAL_MEM_RW that is a demand Memory-write operation.

`0x8233`, **LOCAL_MEM_RW, Access to memory attached to this device, demand access**

The counter counts each access counted by LOCAL_MEM that is a demand Memory-read operation or demand Memory-write operation.

It is IMPLEMENTATION DEFINED whether an access to external memory due to a prefetch to a cache is counted by LOCAL_MEM_RW or LOCAL_MEM_PRFM.

`0x8234`, **LOCAL_MEM_PRFM, Access to memory attached to this device, software prefetch**

The counter counts each access counted by LOCAL_MEM that is due to a prefetch instruction.

It is IMPLEMENTATION DEFINED whether an access to external memory due to a prefetch to a cache is counted by LOCAL_MEM_RW or LOCAL_MEM_PRFM.

`0x8235`, **LOCAL_MEM_LD_RETIRED, Load instruction architecturally executed, access to memory attached to this device**

The counter counts each architecturally executed instruction counted by LOCAL_MEM_LDST_RETIRED that is a Memory-reading instruction.

`0x8236`, **LOCAL_MEM_ST_RETIRED, Store instruction architecturally executed, access to memory attached to this device**

The counter counts each architecturally executed instruction counted by LOCAL_MEM_LDST_RETIRED that is a Memory-writing instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x8237`, **LOCAL_MEM_LDST_RETIRED, Load or store instruction architecturally executed, access to memory attached to this device**

The counter counts each Memory-reading instruction or Memory-writing instruction architecturally executed counted by LDST_ANY_RETIRED that accessed external memory attached to this device.

`0x8238`, **REMOTE_MEM, Access to memory attached to a remote device**

The counter counts each Memory-read operation or Memory-write operation access counted by REMOTE_ACCESS to external memory attached to a remote device.

The counter also counts any accesses to external memory that are not counted by LOCAL_MEM because the implementation classifies them as remote accesses. For example, accesses to expansion memory connected to the device that has significantly higher latency than the main system memory might be classified as remote accesses. For more information, see LOCAL_MEM.

`0x8239`, **REMOTE_MEM_RD, Access to memory attached to a remote device, demand access, read**

The counter counts each access counted by REMOTE_MEM_RW that is a demand Memory-read operation.

`0x823A`, **REMOTE_MEM_WR, Access to memory attached to a remote device, demand access, write**

The counter counts each access counted by REMOTE_MEM_RW that is a demand Memory-write operation.

`0x823B`, **REMOTE_MEM_RW, Access to memory attached to a remote device, demand access**

The counter counts each access counted by REMOTE_MEM that is a demand Memory-read operation or demand Memory-write operation.

`0x823C`, **REMOTE_MEM_PRFM, Access to memory attached to a remote device, software prefetch**

The counter counts each access counted by REMOTE_MEM that is due to a prefetch instruction.

`0x823D`, **REMOTE_MEM_LD_RETIRED, Load instruction architecturally executed, access to memory attached to a remote device**

The counter counts each architecturally executed instruction counted by REMOTE_MEM_LDST_RETIRED that is a Memory-reading instruction.

`0x823E`, **REMOTE_MEM_ST_RETIRED, Store instruction architecturally executed, access to memory attached to a remote device**

The counter counts each architecturally executed instruction counted by REMOTE_MEM_LDST_RETIRED that is a Memory-writing instruction.

`0x823F`, **REMOTE_MEM_LDST_RETIRED, Load or store instruction architecturally executed, access to memory attached to a remote device**

The counter counts each Memory-reading instruction or Memory-writing instruction architecturally executed counted by LDST_ANY_RETIRED that accessed external memory attached to a remote device.

`0x8240`, **L1I_LFB_HIT_RD, Level 1 instruction cache demand fetch line-fill buffer hit**

The counter counts each demand access counted by L1I_CACHE_HIT_RD that hits a recently fetched line in the Level 1 instruction or unified cache.

That is, the access hits a cache line that is in the process of being loaded into the Level 1 instruction or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

`0x8241`, **L2I_LFB_HIT_RD, Level 2 instruction cache demand fetch line-fill buffer hit**

The counter counts each demand access counted by L2I_CACHE_HIT_RD that hits a recently fetched line in the Level 2 instruction or unified cache.

That is, the access hits a cache line that is in the process of being loaded into the Level 2 instruction or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x8244`**, L1D_LFB_HIT_RD, Level 1 data cache demand access line-fill buffer hit, read**

The counter counts each demand access counted by L1D_CACHE_HIT_RD that hits a recently fetched line in the Level 1 data or unified cache.

That is, the access hits a cache line that is in the process of being loaded into the Level 1 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

`0x8245`**, L2D_LFB_HIT_RD, Level 2 data cache demand access line-fill buffer hit, read**

The counter counts each demand access counted by L2D_CACHE_HIT_RD that hits a recently fetched line in the Level 2 data or unified cache.

That is, the access hits a cache line that is in the process of being loaded into the Level 2 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

`0x8246`**, L3D_LFB_HIT_RD, Level 3 data cache demand access line-fill buffer hit, read**

The counter counts each demand access counted by L3D_CACHE_HIT_RD that hits a recently fetched line in the Level 3 data or unified cache.

That is, the access hits a cache line that is in the process of being loaded into the Level 3 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

`0x8247`**, LL_LFB_HIT_RD, Last level cache demand access line-fill buffer hit, read**

The counter counts each demand access counted by LL_CACHE_HIT_RD that hits a recently fetched line in the Last level cache.

That is, the access hits a cache line that is in the process of being loaded into the Last level cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

`0x8248`**, L1D_LFB_HIT_WR, Level 1 data cache demand access line-fill buffer hit, write**

The counter counts each demand access counted by L1D_CACHE_HIT_WR that hits a recently fetched line in the Level 1 data or unified cache.

That is, the access hits a cache line that is in the process of being loaded into the Level 1 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

`0x8249`**, L2D_LFB_HIT_WR, Level 2 data cache demand access line-fill buffer hit, write**

The counter counts each demand access counted by L2D_CACHE_HIT_WR that hits a recently fetched line in the Level 2 data or unified cache.

That is, the access hits a cache line that is in the process of being loaded into the Level 2 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

`0x824A`**, L3D_LFB_HIT_WR, Level 3 data cache demand access line-fill buffer hit, write**

The counter counts each demand access counted by L3D_CACHE_HIT_WR that hits a recently fetched line in the Level 3 data or unified cache.

That is, the access hits a cache line that is in the process of being loaded into the Level 3 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

`0x824B`**, LL_LFB_HIT_WR, Last level cache demand access line-fill buffer hit, write**

The counter counts each demand access counted by LL_CACHE_HIT_WR that hits a recently fetched line in the Last level cache.

That is, the access hits a cache line that is in the process of being loaded into the Last level cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

`0x824C`**, L1D_LFB_HIT_RW, Level 1 data cache demand access line-fill buffer hit**

The counter counts each demand access counted by L1D_CACHE_HIT_RW that hits a recently fetched line in the Level 1 data or unified cache.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

That is, the access hits a cache line that is in the process of being loaded into the Level 1 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

`0x824D`, **L2D_LFB_HIT_RW, Level 2 data cache demand access line-fill buffer hit**

The counter counts each demand access counted by L2D_CACHE_HIT_RW that hits a recently fetched line in the Level 2 data or unified cache.

That is, the access hits a cache line that is in the process of being loaded into the Level 2 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

`0x824E`, **L3D_LFB_HIT_RW, Level 3 data cache demand access line-fill buffer hit**

The counter counts each demand access counted by L3D_CACHE_HIT_RW that hits a recently fetched line in the Level 3 data or unified cache.

That is, the access hits a cache line that is in the process of being loaded into the Level 3 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

`0x824F`, **LL_LFB_HIT_RW, Last level cache demand access line-fill buffer hit**

The counter counts each demand access counted by LL_CACHE_HIT_RW that hits a recently fetched line in the Last level cache.

That is, the access hits a cache line that is in the process of being loaded into the Last level cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

`0x8250`, **L1I_LFB_HIT_RD_FPRFM, Level 1 instruction cache demand fetch line-fill buffer first hit, recently fetched by software prefetch**

The counter counts each demand fetch line-fill buffer first hit counted by L1I_LFB_HIT_RD_FPRF where the cache line was fetched in response to a prefetch instruction.

That is, the fetch hits a cache line that is in the process of being loaded into the Level 1 instruction or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the L1I_CACHE_REFILL_PRFM event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

`0x8251`, **L2I_LFB_HIT_RD_FPRFM, Level 2 instruction cache demand fetch line-fill buffer first hit, recently fetched by software prefetch**

The counter counts each demand fetch line-fill buffer first hit counted by L2I_LFB_HIT_RD_FPRF where the cache line was fetched in response to a prefetch instruction.

That is, the fetch hits a cache line that is in the process of being loaded into the Level 2 instruction or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the L2I_CACHE_REFILL_PRFM event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

`0x8254`, **L1D_LFB_HIT_RD_FPRFM, Level 1 data cache demand access line-fill buffer first hit, read, recently fetched by software prefetch**

The counter counts each line-fill buffer first hit counted by L1D_LFB_HIT_RW_FPRFM that is due to a demand Memory-read operation.

`0x8255`, **L2D_LFB_HIT_RD_FPRFM, Level 2 data cache demand access line-fill buffer first hit, read, recently fetched by software prefetch**

The counter counts each line-fill buffer first hit counted by L2D_LFB_HIT_RW_FPRFM that is due to a demand Memory-read operation.

`0x8256`, **L3D_LFB_HIT_RD_FPRFM, Level 3 data cache demand access line-fill buffer first hit, read, recently fetched by software prefetch**

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The counter counts each line-fill buffer first hit counted by L3D_LFB_HIT_RW_FPRFM that is due to a demand Memory-read operation.

`0x8257`, **LL_LFB_HIT_RD_FPRFM, Last level cache demand access line-fill buffer first hit, read, recently fetched by software prefetch**

The counter counts each line-fill buffer first hit counted by LL_LFB_HIT_RW_FPRFM that is due to a demand Memory-read operation.

`0x8258`, **L1D_LFB_HIT_WR_FPRFM, Level 1 data cache demand access line-fill buffer first hit, write, recently fetched by software prefetch**

The counter counts each line-fill buffer first hit counted by L1D_LFB_HIT_RW_FPRFM that is due to a demand Memory-write operation.

`0x8259`, **L2D_LFB_HIT_WR_FPRFM, Level 2 data cache demand access line-fill buffer first hit, write, recently fetched by software prefetch**

The counter counts each line-fill buffer first hit counted by L2D_LFB_HIT_RW_FPRFM that is due to a demand Memory-write operation.

`0x825A`, **L3D_LFB_HIT_WR_FPRFM, Level 3 data cache demand access line-fill buffer first hit, write, recently fetched by software prefetch**

The counter counts each line-fill buffer first hit counted by L3D_LFB_HIT_RW_FPRFM that is due to a demand Memory-write operation.

`0x825B`, **LL_LFB_HIT_WR_FPRFM, Last level cache demand access line-fill buffer first hit, write, recently fetched by software prefetch**

The counter counts each line-fill buffer first hit counted by LL_LFB_HIT_RW_FPRFM that is due to a demand Memory-write operation.

`0x825C`, **L1D_LFB_HIT_RW_FPRFM, Level 1 data cache demand access line-fill buffer first hit, recently fetched by software prefetch**

The counter counts each demand access line-fill buffer first hit counted by L1D_LFB_HIT_RW_FPRF where the cache line was fetched in response to a prefetch instruction.

That is, the access hits a cache line that is in the process of being loaded into the Level 1 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the L1D_CACHE_REFILL_PRFM event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

`0x825D`, **L2D_LFB_HIT_RW_FPRFM, Level 2 data cache demand access line-fill buffer first hit, recently fetched by software prefetch**

The counter counts each demand access line-fill buffer first hit counted by L2D_LFB_HIT_RW_FPRF where the cache line was fetched in response to a prefetch instruction.

That is, the access hits a cache line that is in the process of being loaded into the Level 2 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the L2D_CACHE_REFILL_PRFM event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

`0x825E`, **L3D_LFB_HIT_RW_FPRFM, Level 3 data cache demand access line-fill buffer first hit, recently fetched by software prefetch**

The counter counts each demand access line-fill buffer first hit counted by L3D_LFB_HIT_RW_FPRF where the cache line was fetched in response to a prefetch instruction.

That is, the access hits a cache line that is in the process of being loaded into the Level 3 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the L3D_CACHE_REFILL_PRFM event was generated when the cache line was fetched into the cache.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

`0x825F`, **LL_LFB_HIT_RW_FPRFM, Last level cache demand access line-fill buffer first hit, recently fetched by software prefetch**

The counter counts each demand access line-fill buffer first hit counted by LL_LFB_HIT_RW_FPRF where the cache line was fetched in response to a prefetch instruction.

That is, the access hits a cache line that is in the process of being loaded into the Last level cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the LL_CACHE_REFILL_PRFM event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

`0x8260`, **L1I_LFB_HIT_RD_FHWPRF, Level 1 instruction cache demand fetch line-fill buffer first hit, recently fetched by hardware prefetcher**

The counter counts each demand fetch line-fill buffer first hit counted by L1I_LFB_HIT_RD_FPRF where the cache line was fetched by a hardware prefetcher.

That is, the fetch hits a cache line that is in the process of being loaded into the Level 1 instruction or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the L1I_CACHE_REFILL_HWPRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

`0x8261`, **L2I_LFB_HIT_RD_FHWPRF, Level 2 instruction cache demand fetch line-fill buffer first hit, recently fetched by hardware prefetcher**

The counter counts each demand fetch line-fill buffer first hit counted by L2I_LFB_HIT_RD_FPRF where the cache line was fetched by a hardware prefetcher.

That is, the fetch hits a cache line that is in the process of being loaded into the Level 2 instruction or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the L2I_CACHE_REFILL_HWPRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

`0x8264`, **L1D_LFB_HIT_RD_FHWPRF, Level 1 data cache demand access line-fill buffer first hit, read, recently fetched by hardware prefetcher**

The counter counts each line-fill buffer first hit counted by L1D_LFB_HIT_RW_FHWPRF that is due to a demand Memory-read operation.

`0x8265`, **L2D_LFB_HIT_RD_FHWPRF, Level 2 data cache demand access line-fill buffer first hit, read, recently fetched by hardware prefetcher**

The counter counts each line-fill buffer first hit counted by L2D_LFB_HIT_RW_FHWPRF that is due to a demand Memory-read operation.

`0x8266`, **L3D_LFB_HIT_RD_FHWPRF, Level 3 data cache demand access line-fill buffer first hit, read, recently fetched by hardware prefetcher**

The counter counts each line-fill buffer first hit counted by L3D_LFB_HIT_RW_FHWPRF that is due to a demand Memory-read operation.

`0x8267`, **LL_LFB_HIT_RD_FHWPRF, Last level cache demand access line-fill buffer first hit, read, recently fetched by hardware prefetcher**

The counter counts each line-fill buffer first hit counted by LL_LFB_HIT_RW_FHWPRF that is due to a demand Memory-read operation.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x8268`, **L1D_LFB_HIT_WR_FHWPRF, Level 1 data cache demand access line-fill buffer first hit, write, recently fetched by hardware prefetcher**

The counter counts each line-fill buffer first hit counted by L1D_LFB_HIT_RW_FHWPRF that is due to a demand Memory-write operation.

`0x8269`, **L2D_LFB_HIT_WR_FHWPRF, Level 2 data cache demand access line-fill buffer first hit, write, recently fetched by hardware prefetcher**

The counter counts each line-fill buffer first hit counted by L2D_LFB_HIT_RW_FHWPRF that is due to a demand Memory-write operation.

`0x826A`, **L3D_LFB_HIT_WR_FHWPRF, Level 3 data cache demand access line-fill buffer first hit, write, recently fetched by hardware prefetcher**

The counter counts each line-fill buffer first hit counted by L3D_LFB_HIT_RW_FHWPRF that is due to a demand Memory-write operation.

`0x826B`, **LL_LFB_HIT_WR_FHWPRF, Last level cache demand access line-fill buffer first hit, write, recently fetched by hardware prefetcher**

The counter counts each line-fill buffer first hit counted by LL_LFB_HIT_RW_FHWPRF that is due to a demand Memory-write operation.

`0x826C`, **L1D_LFB_HIT_RW_FHWPRF, Level 1 data cache demand access line-fill buffer first hit, recently fetched by hardware prefetcher**

The counter counts each demand access line-fill buffer first hit counted by L1D_LFB_HIT_RW_FPRF where the cache line was fetched by a hardware prefetcher.

That is, the access hits a cache line that is in the process of being loaded into the Level 1 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the L1D_CACHE_REFILL_HWPRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

`0x826D`, **L2D_LFB_HIT_RW_FHWPRF, Level 2 data cache demand access line-fill buffer first hit, recently fetched by hardware prefetcher**

The counter counts each demand access line-fill buffer first hit counted by L2D_LFB_HIT_RW_FPRF where the cache line was fetched by a hardware prefetcher.

That is, the access hits a cache line that is in the process of being loaded into the Level 2 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the L2D_CACHE_REFILL_HWPRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

`0x826E`, **L3D_LFB_HIT_RW_FHWPRF, Level 3 data cache demand access line-fill buffer first hit, recently fetched by hardware prefetcher**

The counter counts each demand access line-fill buffer first hit counted by L3D_LFB_HIT_RW_FPRF where the cache line was fetched by a hardware prefetcher.

That is, the access hits a cache line that is in the process of being loaded into the Level 3 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the L3D_CACHE_REFILL_HWPRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

`0x826F`, **LL_LFB_HIT_RW_FHWPRF, Last level cache demand access line-fill buffer first hit, recently fetched by hardware prefetcher**

The counter counts each demand access line-fill buffer first hit counted by LL_LFB_HIT_RW_FPRF where the cache line was fetched by a hardware prefetcher.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

That is, the access hits a cache line that is in the process of being loaded into the Last level cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the LL_CACHE_REFILL_HWPRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

`0x8270`, **L1I_LFB_HIT_RD_FPRF, Level 1 instruction cache demand fetch line-fill buffer first hit, recently fetched by prefetch**

The counter counts each demand fetch line-fill buffer first hit counted by L1I_LFB_HIT_RD where the cache line was fetched in response to a prefetch instruction or by a hardware prefetcher.

That is, the fetch hits a cache line that is in the process of being loaded into the Level 1 instruction or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the L1I_CACHE_REFILL_PRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

`0x8271`, **L2I_LFB_HIT_RD_FPRF, Level 2 instruction cache demand fetch line-fill buffer first hit, recently fetched by prefetch**

The counter counts each demand fetch line-fill buffer first hit counted by L2I_LFB_HIT_RD where the cache line was fetched in response to a prefetch instruction or by a hardware prefetcher.

That is, the fetch hits a cache line that is in the process of being loaded into the Level 2 instruction or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the L2I_CACHE_REFILL_PRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

`0x8274`, **L1D_LFB_HIT_RD_FPRF, Level 1 data cache demand access line-fill buffer first hit, read, recently fetched by prefetch**

The counter counts each line-fill buffer first hit counted by L1D_LFB_HIT_RW_FPRF that is due to a demand Memory-read operation.

`0x8275`, **L2D_LFB_HIT_RD_FPRF, Level 2 data cache demand access line-fill buffer first hit, read, recently fetched by prefetch**

The counter counts each line-fill buffer first hit counted by L2D_LFB_HIT_RW_FPRF that is due to a demand Memory-read operation.

`0x8276`, **L3D_LFB_HIT_RD_FPRF, Level 3 data cache demand access line-fill buffer first hit, read, recently fetched by prefetch**

The counter counts each line-fill buffer first hit counted by L3D_LFB_HIT_RW_FPRF that is due to a demand Memory-read operation.

`0x8277`, **LL_LFB_HIT_RD_FPRF, Last level cache demand access line-fill buffer first hit, read, recently fetched by prefetch**

The counter counts each line-fill buffer first hit counted by LL_LFB_HIT_RW_FPRF that is due to a demand Memory-read operation.

`0x8278`, **L1D_LFB_HIT_WR_FPRF, Level 1 data cache demand access line-fill buffer first hit, write, recently fetched by prefetch**

The counter counts each line-fill buffer first hit counted by L1D_LFB_HIT_RW_FPRF that is due to a demand Memory-write operation.

`0x8279`, **L2D_LFB_HIT_WR_FPRF, Level 2 data cache demand access line-fill buffer first hit, write, recently fetched by prefetch**

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The counter counts each line-fill buffer first hit counted by L2D_LFB_HIT_RW_FPRF that is due to a demand Memory-write operation.

`0x827A`, **L3D_LFB_HIT_WR_FPRF, Level 3 data cache demand access line-fill buffer first hit, write, recently fetched by prefetch**

The counter counts each line-fill buffer first hit counted by L3D_LFB_HIT_RW_FPRF that is due to a demand Memory-write operation.

`0x827B`, **LL_LFB_HIT_WR_FPRF, Last level cache demand access line-fill buffer first hit, write, recently fetched by prefetch**

The counter counts each line-fill buffer first hit counted by LL_LFB_HIT_RW_FPRF that is due to a demand Memory-write operation.

`0x827C`, **L1D_LFB_HIT_RW_FPRF, Level 1 data cache demand access line-fill buffer first hit, recently fetched by prefetch**

The counter counts each demand access line-fill buffer first hit counted by L1D_LFB_HIT_RW where the cache line was fetched in response to a prefetch instruction or by a hardware prefetcher.

That is, the access hits a cache line that is in the process of being loaded into the Level 1 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the L1D_CACHE_REFILL_PRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

`0x827D`, **L2D_LFB_HIT_RW_FPRF, Level 2 data cache demand access line-fill buffer first hit, recently fetched by prefetch**

The counter counts each demand access line-fill buffer first hit counted by L2D_LFB_HIT_RW where the cache line was fetched in response to a prefetch instruction or by a hardware prefetcher.

That is, the access hits a cache line that is in the process of being loaded into the Level 2 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the L2D_CACHE_REFILL_PRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

`0x827E`, **L3D_LFB_HIT_RW_FPRF, Level 3 data cache demand access line-fill buffer first hit, recently fetched by prefetch**

The counter counts each demand access line-fill buffer first hit counted by L3D_LFB_HIT_RW where the cache line was fetched in response to a prefetch instruction or by a hardware prefetcher.

That is, the access hits a cache line that is in the process of being loaded into the Level 3 data or unified cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the L3D_CACHE_REFILL_PRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

`0x827F`, **LL_LFB_HIT_RW_FPRF, Last level cache demand access line-fill buffer first hit, recently fetched by prefetch**

The counter counts each demand access line-fill buffer first hit counted by LL_LFB_HIT_RW where the cache line was fetched in response to a prefetch instruction or by a hardware prefetcher.

That is, the access hits a cache line that is in the process of being loaded into the Last level cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the LL_CACHE_REFILL_PRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x8280`**, L1I_CACHE_PRF, Level 1 instruction cache, prefetch hit**

The counter counts each access counted by L1I_CACHE that is due to a prefetch instruction, or hardware prefetch.

`0x8281`**, L2I_CACHE_PRF, Level 2 instruction cache, prefetch hit**

The counter counts each access counted by L2I_CACHE that is due to a prefetch instruction, or hardware prefetch.

This includes accesses to the Level 2 instruction or unified cache due to a refill of another cache caused by a prefetch instruction, or hardware prefetch.

`0x8284`**, L1D_CACHE_PRF, Level 1 data cache, prefetch hit**

The counter counts each access counted by L1D_CACHE that is due to a prefetch instruction, or hardware prefetch.

`0x8285`**, L2D_CACHE_PRF, Level 2 data cache, prefetch hit**

The counter counts each access counted by L2D_CACHE that is due to a prefetch instruction, or hardware prefetch.

This includes accesses to the Level 2 data or unified cache due to a refill of another cache caused by a prefetch instruction, or hardware prefetch.

`0x8286`**, L3D_CACHE_PRF, Level 3 data cache, prefetch hit**

The counter counts each access counted by L3D_CACHE that is due to a prefetch instruction, or hardware prefetch.

This includes accesses to the Level 3 data or unified cache due to a refill of another cache caused by a prefetch instruction, or hardware prefetch.

`0x8287`**, LL_CACHE_PRF, Last level cache, prefetch hit**

The counter counts each access counted by LL_CACHE that is due to a prefetch instruction, or hardware prefetch.

This includes accesses to the Last level cache due to a refill of another cache caused by a prefetch instruction, or hardware prefetch.

`0x8288`**, L1I_CACHE_REFILL_PRF, Level 1 instruction cache refill, prefetch hit**

The counter counts each access counted by L1I_CACHE_PRF that causes a refill of the Level 1 instruction or unified cache from outside of the Level 1 instruction or unified cache.

`0x8289`**, L2I_CACHE_REFILL_PRF, Level 2 instruction cache refill, prefetch hit**

The counter counts each access counted by L2I_CACHE_PRF that causes a refill of the Level 2 instruction or unified cache, or any Level 1 data, instruction, or unified cache of this PE, from outside of those caches.

`0x828C`**, L1D_CACHE_REFILL_PRF, Level 1 data cache refill, prefetch hit**

The counter counts each access counted by L1D_CACHE_PRF that causes a refill of the Level 1 data or unified cache from outside of the Level 1 data or unified cache.

`0x828D`**, L2D_CACHE_REFILL_PRF, Level 2 data cache refill, prefetch hit**

The counter counts each access counted by L2D_CACHE_PRF that causes a refill of the Level 2 data or unified cache, or any Level 1 data, instruction, or unified cache of this PE, from outside of those caches.

`0x828E`**, L3D_CACHE_REFILL_PRF, Level 3 data cache refill, prefetch hit**

The counter counts each access counted by L3D_CACHE_PRF that causes a refill of the Level 3 data or unified cache, or any Level 1 or Level 2 data, instruction, or unified cache of this PE, from outside of those caches.

`0x828F`**, LL_CACHE_REFILL_PRF, Last level cache refill, prefetch hit**

The counter counts each access counted by LL_CACHE_PRF that causes a refill of the Last level cache, or any other data, instruction, or unified cache of this PE, from outside of those caches.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x8290`**, ISNP_HIT_PRF, Snoop hit, instruction prefetch**

The counter counts each snoop generated by the PE in response to an instruction prefetch instruction, or by an instruction hardware prefetch that hits in a cache outside of the cache hierarchy of this PE.

The event is counted by the PE generating the snoop, not the PE being snooped.

`0x8291`**, ISNP_HIT_NEAR_PRF, Snoop hit in near local cache, instruction prefetch**

The counter counts each snoop generated by the PE in response to an instruction prefetch instruction, or by an instruction hardware prefetch counted by ISNP_HIT_PRF that hits in a cache outside of the cache hierarchy of this PE in the local PE cluster.

The event is counted by the PE generating the snoop, not the PE being snooped.

`0x8292`**, ISNP_HIT_FAR_PRF, Snoop hit in far local cache, instruction prefetch**

The counter counts each snoop generated by the PE in response to an instruction prefetch instruction, or by an instruction hardware prefetch counted by ISNP_HIT_PRF that hits in a cache outside the local PE cluster on the same device.

The event is counted by the PE generating the snoop, not the PE being snooped.

`0x8293`**, ISNP_HIT_REMOTE_PRF, Snoop hit in remote cache, instruction prefetch**

The counter counts each snoop generated by the PE in response to an instruction prefetch instruction, or by an instruction hardware prefetch counted by ISNP_HIT_PRF that hits in a cache on a remote device.

The event is counted by the PE generating the snoop, not the PE being snooped.

`0x8294`**, DSNP_HIT_PRF, Snoop hit, data prefetch**

The counter counts each snoop generated by the PE in response to a data prefetch instruction, or by a data hardware prefetch that hits in a cache outside of the cache hierarchy of this PE.

The event is counted by the PE generating the snoop, not the PE being snooped.

`0x8295`**, DSNP_HIT_NEAR_PRF, Snoop hit in near local cache, data prefetch**

The counter counts each snoop generated by the PE in response to a data prefetch instruction, or by a data hardware prefetch counted by DSNP_HIT_PRF that hits in a cache outside of the cache hierarchy of this PE in the local PE cluster.

The event is counted by the PE generating the snoop, not the PE being snooped.

`0x8296`**, DSNP_HIT_FAR_PRF, Snoop hit in far local cache, data prefetch**

The counter counts each snoop generated by the PE in response to a data prefetch instruction, or by a data hardware prefetch counted by DSNP_HIT_PRF that hits in a cache outside the local PE cluster on the same device.

The event is counted by the PE generating the snoop, not the PE being snooped.

`0x8297`**, DSNP_HIT_REMOTE_PRF, Snoop hit in remote cache, data prefetch**

The counter counts each snoop generated by the PE in response to a data prefetch instruction, or by a data hardware prefetch counted by DSNP_HIT_PRF that hits in a cache on a remote device.

The event is counted by the PE generating the snoop, not the PE being snooped.

`0x8298`**, LL_CACHE_RW, Last level cache demand access**

The counter counts each access counted by LL_CACHE that is due to a demand Memory-read operation or demand Memory-write operation.

This includes:

- Accesses made by speculatively executed instructions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Accesses to the Last level cache due to a refill of another cache caused by a demand Memory-read operation or demand Memory-write operation.

Arm recommends that this event is implemented if any of the following are true:

- Event LL_CACHE_PRFM is implemented.
- Event LL_CACHE_HWPRF is implemented.

`0x8299`, **LL_CACHE_PRFM, Last level cache software prefetch**

The counter counts each access counted by LL_CACHE_PRF that is due to a prefetch instruction.

This includes accesses to the Last level cache due to a refill of another cache caused by a prefetch instruction.

Arm recommends that this event is implemented if event LL_CACHE_RW is implemented.

`0x829A`, **LL_CACHE_REFILL, Last level cache refill**

The counter counts each access counted by LL_CACHE that causes a refill of the Last level cache, or any other data, instruction, or unified cache of this PE, from outside of those caches.

A refill includes any access that causes data to be fetched from outside of the Level 1 to Last level caches, even if the data is ultimately not allocated into the Last level cache. For example, data might be fetched into a buffer but then discarded, rather than being allocated into a cache. These buffers are treated as part of the cache.

For example, the counter counts:

- Accesses to the Last level cache that cause a refill that is satisfier by fetching data from memory, or a Last level cache of another PE.
- Refills of and write-backs from any other data, instruction, or unified caches of this PE that cause a refill from outside of the Level 1 to Last level caches of this PE.
- Accesses to the Last level cache that cause a refill of a higher level cache of this PE from outside of the Level 1 to Last level caches of this PE, even if there is no refill of the Last level cache.

The counter does not count accesses that:

- Miss in the cache but are satisfied by the refill of a previous miss and do not cause a new refill, even if that previous refill is not complete at the time of the miss.
- Miss in the cache but do not generate a refill, such as a write through the cache.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x829B`, **LL_CACHE_REFILL_PRFM, Last level cache refill, software prefetch**

The counter counts each access counted by LL_CACHE_PRFM that causes a refill of the Last level cache, or any other data, instruction, or unified cache of this PE, from outside of those caches.

`0x829C`, **LL_CACHE_WB, Last level cache write-back**

The counter counts each write-back of data from the Last level cache to outside of the Level 1 to Last level caches.

For example:

- A write-back of a dirty cache line to memory.
- A write-back of a recently fetched cache line that has not been allocated to the Last level cache.

Each write-back is counted once, even if multiple accesses are required to complete the write-back.

It is IMPLEMENTATION DEFINED whether the counter counts:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- A transfer of a dirty cache line from the Last level cache to outside of Level 1 to Last level caches made as a result of a coherency request.
- Write-backs made as a result of cache maintenance instructions.

The counter does not count:

- The invalidation of a cache line without any write-back to outside of the Level 1 to Last level caches or memory.
- Writes that write through the Last level cache to outside of the Level 1 to Last level caches.
- Transfers of data from the Last level cache to another cache to satisfy a refill of the other cache.

A write-back is attributable to the agent that generated the request that caused the write-back. This might not be the same agent that caused the data being written back to be allocated into the cache.

An Unattributable write-back event occurs when a requestor outside of the PE makes a coherency request that results in write-back. If the cache is shared, then an Unattributable write-back event is not counted. If the cache is not shared, then the event is counted.

It is IMPLEMENTATION DEFINED whether a write of a whole cache line that is not the result of the eviction of a line from the cache, is counted. For example, this applies when the PE determines software is streaming writes to memory and does not allocate lines to the cache, or by a `DC ZVA` operation.

`0x829D`, **LL_CACHE_WR, Last level cache access, write**

If the LL_CACHE_RW event is implemented, the counter counts each access counted by LL_CACHE_RW that is a Memory-write operation.

If the LL_CACHE_RW event is not implemented, the counter counts each access counted by LL_CACHE that is a Memory-write operation.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x829F`, **LL_CACHE_REFILL_WR, Last level cache refill, write**

The counter counts each access counted by both LL_CACHE_REFILL and LL_CACHE_WR.

That is, every refill of the Last level cache counted by LL_CACHE_REFILL that is caused by a Memory-write operation.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x82A0`, **MEM_ACCESS_RW, Data memory access, demand access**

The counter counts each access counted by MEM_ACCESS that is a demand Memory-read operation or demand Memory-write operation.

This includes accesses made by speculative instructions.

`0x82A1`, **INST_FETCH_RD, Instruction memory access, demand fetch**

The counter counts each fetch counted by INST_FETCH that is a demand Instruction memory access.

This includes any fetch made for a speculative instruction.

`0x82A2`, **MEM_ACCESS_PRFM, Data memory access, software prefetch**

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The counter counts each access counted by MEM_ACCESS that is due to a prefetch instruction.

`0x82A3`, **INST_FETCH_PRFM, Instruction memory access, software prefetch**

The counter counts each fetch counted by INST_FETCH that is due to a prefetch instruction.

`0x82B0`, **L1I_CACHE_MISS_RETIRED, Instruction architecturally executed, miss in Level 1 instruction cache**

The counter counts each instruction architecturally executed counted by INST_RETIRED that missed in Level 1 instruction or unified cache when fetched.

`0x82B1`, **L2I_CACHE_MISS_RETIRED, Instruction architecturally executed, miss in Level 2 instruction cache**

The counter counts each instruction architecturally executed counted by INST_RETIRED that missed in Level 2 instruction or unified cache when fetched.

`0x82B4`, **L1D_CACHE_MISS_LD_RETIRED, Load instruction architecturally executed, miss in Level 1 data cache**

The counter counts each architecturally executed instruction counted by L1D_CACHE_MISS_LDST_RETIRED that is a Memory-reading instruction.

`0x82B5`, **L2D_CACHE_MISS_LD_RETIRED, Load instruction architecturally executed, miss in Level 2 data cache**

The counter counts each architecturally executed instruction counted by L2D_CACHE_MISS_LDST_RETIRED that is a Memory-reading instruction.

`0x82B6`, **L3D_CACHE_MISS_LD_RETIRED, Load instruction architecturally executed, miss in Level 3 data cache**

The counter counts each architecturally executed instruction counted by L3D_CACHE_MISS_LDST_RETIRED that is a Memory-reading instruction.

`0x82B7`, **LL_CACHE_MISS_LD_RETIRED, Load instruction architecturally executed, miss in Last level cache**

The counter counts each architecturally executed instruction counted by LL_CACHE_MISS_LDST_RETIRED that is a Memory-reading instruction.

`0x82B8`, **L1D_CACHE_MISS_ST_RETIRED, Store instruction architecturally executed, miss in Level 1 data cache**

The counter counts each architecturally executed instruction counted by L1D_CACHE_MISS_LDST_RETIRED that is a Memory-writing instruction.

`0x82B9`, **L2D_CACHE_MISS_ST_RETIRED, Store instruction architecturally executed, miss in Level 2 data cache**

The counter counts each architecturally executed instruction counted by L2D_CACHE_MISS_LDST_RETIRED that is a Memory-writing instruction.

`0x82BA`, **L3D_CACHE_MISS_ST_RETIRED, Store instruction architecturally executed, miss in Level 3 data cache**

The counter counts each architecturally executed instruction counted by L3D_CACHE_MISS_LDST_RETIRED that is a Memory-writing instruction.

`0x82BB`, **LL_CACHE_MISS_ST_RETIRED, Store instruction architecturally executed, miss in Last level cache**

The counter counts each architecturally executed instruction counted by LL_CACHE_MISS_LDST_RETIRED that is a Memory-writing instruction.

`0x82BC`, **L1D_CACHE_MISS_LDST_RETIRED, Load or store instruction architecturally executed, miss in Level 1 data cache**

The counter counts each architecturally executed Memory-reading instruction or Memory-writing instruction counted by LDST_ANY_RETIRED that missed in Level 1 data or unified cache.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x82BD`, **L2D_CACHE_MISS_LDST_RETIRED, Load or store instruction architecturally executed, miss in Level 2 data cache**

The counter counts each architecturally executed Memory-reading instruction or Memory-writing instruction counted by LDST_ANY_RETIRED that missed in Level 2 data or unified cache.

`0x82BE`, **L3D_CACHE_MISS_LDST_RETIRED, Load or store instruction architecturally executed, miss in Level 3 data cache**

The counter counts each architecturally executed Memory-reading instruction or Memory-writing instruction counted by LDST_ANY_RETIRED that missed in Level 3 data or unified cache.

`0x82BF`, **LL_CACHE_MISS_LDST_RETIRED, Load or store instruction architecturally executed, miss in Last level cache**

The counter counts each architecturally executed Memory-reading instruction or Memory-writing instruction counted by LDST_ANY_RETIRED that missed in Last level cache.

`0x82C4`, **L1D_CACHE_HITM_LD_RETIRED, Load instruction architecturally executed, hit modified data in Level 1 data cache**

The counter counts each architecturally executed instruction counted by L1D_CACHE_HITM_LDST_RETIRED that is a Memory-reading instruction.

`0x82C5`, **L2D_CACHE_HITM_LD_RETIRED, Load instruction architecturally executed, hit modified data in Level 2 data cache**

The counter counts each architecturally executed instruction counted by L2D_CACHE_HITM_LDST_RETIRED that is a Memory-reading instruction.

`0x82C6`, **L3D_CACHE_HITM_LD_RETIRED, Load instruction architecturally executed, hit modified data in Level 3 data cache**

The counter counts each architecturally executed instruction counted by L3D_CACHE_HITM_LDST_RETIRED that is a Memory-reading instruction.

`0x82C7`, **LL_CACHE_HITM_LD_RETIRED, Load instruction architecturally executed, hit modified data in Last level cache**

The counter counts each architecturally executed instruction counted by LL_CACHE_HITM_LDST_RETIRED that is a Memory-reading instruction.

`0x82C8`, **L1D_CACHE_HITM_ST_RETIRED, Store instruction architecturally executed, hit modified data in Level 1 data cache**

The counter counts each architecturally executed instruction counted by L1D_CACHE_HITM_LDST_RETIRED that is a Memory-writing instruction.

`0x82C9`, **L2D_CACHE_HITM_ST_RETIRED, Store instruction architecturally executed, hit modified data in Level 2 data cache**

The counter counts each architecturally executed instruction counted by L2D_CACHE_HITM_LDST_RETIRED that is a Memory-writing instruction.

`0x82CA`, **L3D_CACHE_HITM_ST_RETIRED, Store instruction architecturally executed, hit modified data in Level 3 data cache**

The counter counts each architecturally executed instruction counted by L3D_CACHE_HITM_LDST_RETIRED that is a Memory-writing instruction.

`0x82CB`, **LL_CACHE_HITM_ST_RETIRED, Store instruction architecturally executed, hit modified data in Last level cache**

The counter counts each architecturally executed instruction counted by LL_CACHE_HITM_LDST_RETIRED that is a Memory-writing instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x82CC`, **L1D_CACHE_HITM_LDST_RETIRED, Load or store instruction architecturally executed, hit modified data in Level 1 data cache**

The counter counts each architecturally executed Memory-reading instruction or Memory-writing instruction counted by L1D_CACHE_HIT_LDST_RETIRED that hit modified data in Level 1 data or unified cache.

`0x82CD`, **L2D_CACHE_HITM_LDST_RETIRED, Load or store instruction architecturally executed, hit modified data in Level 2 data cache**

The counter counts each architecturally executed Memory-reading instruction or Memory-writing instruction counted by L2D_CACHE_HIT_LDST_RETIRED that hit modified data in Level 2 data or unified cache.

`0x82CE`, **L3D_CACHE_HITM_LDST_RETIRED, Load or store instruction architecturally executed, hit modified data in Level 3 data cache**

The counter counts each architecturally executed Memory-reading instruction or Memory-writing instruction counted by L3D_CACHE_HIT_LDST_RETIRED that hit modified data in Level 3 data or unified cache.

`0x82CF`, **LL_CACHE_HITM_LDST_RETIRED, Load or store instruction architecturally executed, hit modified data in Last level cache**

The counter counts each architecturally executed Memory-reading instruction or Memory-writing instruction counted by LL_CACHE_HIT_LDST_RETIRED that hit modified data in Last level cache.

`0x82D0`, **L1I_LFB_HIT_RETIRED, Instruction architecturally executed, line-fill buffer hit in Level 1 instruction cache**

The counter counts each instruction architecturally executed counted by L1I_CACHE_HIT_RETIRED that hit a recently fetched line in Level 1 instruction or unified cache when fetched.

`0x82D1`, **L2I_LFB_HIT_RETIRED, Instruction architecturally executed, line-fill buffer hit in Level 2 instruction cache**

The counter counts each instruction architecturally executed counted by L2I_CACHE_HIT_RETIRED that hit a recently fetched line in Level 2 instruction or unified cache when fetched.

`0x82D4`, **L1D_LFB_HIT_LD_RETIRED, Load instruction architecturally executed, line-fill buffer hit in Level 1 data cache**

The counter counts each architecturally executed instruction counted by L1D_LFB_HIT_LDST_RETIRED that is a Memory-reading instruction.

`0x82D5`, **L2D_LFB_HIT_LD_RETIRED, Load instruction architecturally executed, line-fill buffer hit in Level 2 data cache**

The counter counts each architecturally executed instruction counted by L2D_LFB_HIT_LDST_RETIRED that is a Memory-reading instruction.

`0x82D6`, **L3D_LFB_HIT_LD_RETIRED, Load instruction architecturally executed, line-fill buffer hit in Level 3 data cache**

The counter counts each architecturally executed instruction counted by L3D_LFB_HIT_LDST_RETIRED that is a Memory-reading instruction.

`0x82D7`, **LL_LFB_HIT_LD_RETIRED, Load instruction architecturally executed, line-fill buffer hit in Last level cache**

The counter counts each architecturally executed instruction counted by LL_LFB_HIT_LDST_RETIRED that is a Memory-reading instruction.

`0x82D8`, **L1D_LFB_HIT_ST_RETIRED, Store instruction architecturally executed, line-fill buffer hit in Level 1 data cache**

The counter counts each architecturally executed instruction counted by L1D_LFB_HIT_LDST_RETIRED that is a Memory-writing instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x82D9`, **L2D_LFB_HIT_ST_RETIRED, Store instruction architecturally executed, line-fill buffer hit in Level 2 data cache**

The counter counts each architecturally executed instruction counted by L2D_LFB_HIT_LDST_RETIRED that is a Memory-writing instruction.

`0x82DA`, **L3D_LFB_HIT_ST_RETIRED, Store instruction architecturally executed, line-fill buffer hit in Level 3 data cache**

The counter counts each architecturally executed instruction counted by L3D_LFB_HIT_LDST_RETIRED that is a Memory-writing instruction.

`0x82DB`, **LL_LFB_HIT_ST_RETIRED, Store instruction architecturally executed, line-fill buffer hit in Last level cache**

The counter counts each architecturally executed instruction counted by LL_LFB_HIT_LDST_RETIRED that is a Memory-writing instruction.

`0x82DC`, **L1D_LFB_HIT_LDST_RETIRED, Load or store instruction architecturally executed, line-fill buffer hit in Level 1 data cache**

The counter counts each architecturally executed Memory-reading instruction or Memory-writing instruction counted by L1D_CACHE_HIT_LDST_RETIRED that hit a recently fetched line in Level 1 data or unified cache.

`0x82DD`, **L2D_LFB_HIT_LDST_RETIRED, Load or store instruction architecturally executed, line-fill buffer hit in Level 2 data cache**

The counter counts each architecturally executed Memory-reading instruction or Memory-writing instruction counted by L2D_CACHE_HIT_LDST_RETIRED that hit a recently fetched line in Level 2 data or unified cache.

`0x82DE`, **L3D_LFB_HIT_LDST_RETIRED, Load or store instruction architecturally executed, line-fill buffer hit in Level 3 data cache**

The counter counts each architecturally executed Memory-reading instruction or Memory-writing instruction counted by L3D_CACHE_HIT_LDST_RETIRED that hit a recently fetched line in Level 3 data or unified cache.

`0x82DF`, **LL_LFB_HIT_LDST_RETIRED, Load or store instruction architecturally executed, line-fill buffer hit in Last level cache**

The counter counts each architecturally executed Memory-reading instruction or Memory-writing instruction counted by LL_CACHE_HIT_LDST_RETIRED that hit a recently fetched line in Last level cache.

`0x82E0`, **L1I_CACHE_HIT_RETIRED, Instruction architecturally executed, hit in Level 1 instruction cache**

The counter counts each instruction architecturally executed counted by INST_RETIRED that hit in Level 1 instruction or unified cache when fetched.

`0x82E1`, **L2I_CACHE_HIT_RETIRED, Instruction architecturally executed, hit in Level 2 instruction cache**

The counter counts each instruction architecturally executed counted by INST_RETIRED that hit in Level 2 instruction or unified cache when fetched.

`0x82E4`, **L1D_CACHE_HIT_LD_RETIRED, Load instruction architecturally executed, hit in Level 1 data cache**

The counter counts each architecturally executed instruction counted by L1D_CACHE_HIT_LDST_RETIRED that is a Memory-reading instruction.

`0x82E5`, **L2D_CACHE_HIT_LD_RETIRED, Load instruction architecturally executed, hit in Level 2 data cache**

The counter counts each architecturally executed instruction counted by L2D_CACHE_HIT_LDST_RETIRED that is a Memory-reading instruction.

`0x82E6`, **L3D_CACHE_HIT_LD_RETIRED, Load instruction architecturally executed, hit in Level 3 data cache**

The counter counts each architecturally executed instruction counted by L3D_CACHE_HIT_LDST_RETIRED that is a Memory-reading instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x82E7`, **LL_CACHE_HIT_LD_RETIRED, Load instruction architecturally executed, hit in Last level cache**

The counter counts each architecturally executed instruction counted by LL_CACHE_HIT_LDST_RETIRED that is a Memory-reading instruction.

`0x82E8`, **L1D_CACHE_HIT_ST_RETIRED, Store instruction architecturally executed, hit in Level 1 data cache**

The counter counts each architecturally executed instruction counted by L1D_CACHE_HIT_LDST_RETIRED that is a Memory-writing instruction.

`0x82E9`, **L2D_CACHE_HIT_ST_RETIRED, Store instruction architecturally executed, hit in Level 2 data cache**

The counter counts each architecturally executed instruction counted by L2D_CACHE_HIT_LDST_RETIRED that is a Memory-writing instruction.

`0x82EA`, **L3D_CACHE_HIT_ST_RETIRED, Store instruction architecturally executed, hit in Level 3 data cache**

The counter counts each architecturally executed instruction counted by L3D_CACHE_HIT_LDST_RETIRED that is a Memory-writing instruction.

`0x82EB`, **LL_CACHE_HIT_ST_RETIRED, Store instruction architecturally executed, hit in Last level cache**

The counter counts each architecturally executed instruction counted by LL_CACHE_HIT_LDST_RETIRED that is a Memory-writing instruction.

`0x82EC`, **L1D_CACHE_HIT_LDST_RETIRED, Load or store instruction architecturally executed, hit in Level 1 data cache**

The counter counts each architecturally executed Memory-reading instruction or Memory-writing instruction counted by LDST_ANY_RETIRED that hit in Level 1 data or unified cache.

`0x82ED`, **L2D_CACHE_HIT_LDST_RETIRED, Load or store instruction architecturally executed, hit in Level 2 data cache**

The counter counts each architecturally executed Memory-reading instruction or Memory-writing instruction counted by LDST_ANY_RETIRED that hit in Level 2 data or unified cache.

`0x82EE`, **L3D_CACHE_HIT_LDST_RETIRED, Load or store instruction architecturally executed, hit in Level 3 data cache**

The counter counts each architecturally executed Memory-reading instruction or Memory-writing instruction counted by LDST_ANY_RETIRED that hit in Level 3 data or unified cache.

`0x82EF`, **LL_CACHE_HIT_LDST_RETIRED, Load or store instruction architecturally executed, hit in Last level cache**

The counter counts each architecturally executed Memory-reading instruction or Memory-writing instruction counted by LDST_ANY_RETIRED that hit in Last level cache.

`0x82F0`, **ITLB_HIT_RETIRED, Instruction architecturally executed, no translation table walk**

The counter counts each instruction which was architecturally executed counted by INST_RETIRED that did not cause a refill of an instruction TLB involving at least one translation table walk access.

The counter does not count the instruction which was architecturally executed if any of the following are true:

- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the instruction which was architecturally executed if the access generates a Translation fault for any other reason.

`0x82F1`, **DTLB_HIT_LD_RETIRED, Load instruction architecturally executed, no translation table walk**

The counter counts each Memory-reading instruction architecturally executed counted by DTLB_HIT_LDST_RETIRED that did not cause a refill of a data TLB involving at least one translation table walk access.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x82F2`, **DTLB_HIT_ST_RETIRED, Store instruction architecturally executed, no translation table walk**

The counter counts each Memory-writing instruction architecturally executed counted by DTLB_HIT_LDST_RETIRED that did not cause a refill of a data TLB involving at least one translation table walk access.

`0x82F3`, **DTLB_HIT_LDST_RETIRED, Load or store instruction architecturally executed, no translation table walk**

The counter counts each Memory-reading instruction or Memory-writing instruction architecturally executed counted by LDST_ANY_RETIRED that did not cause a refill of a data TLB involving at least one translation table walk access.

`0x82F4`, **ITLB_WALK_RETIRED, Instruction architecturally executed, at least one translation table walk**

The counter counts each instruction which was architecturally executed counted by INST_RETIRED that caused a refill of an instruction TLB involving at least one translation table walk access.

The counter does not count the instruction which was architecturally executed if any of the following are true:

- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the instruction which was architecturally executed if the access generates a Translation fault for any other reason.

`0x82F5`, **DTLB_WALK_LD_RETIRED, Load instruction architecturally executed, at least one translation table walk**

The counter counts each Memory-reading instruction architecturally executed counted by DTLB_WALK_LDST_RETIRED that caused a refill of a data TLB involving at least one translation table walk access.

The counter does not count the Memory-reading instruction architecturally executed if any of the following are true:

- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the Memory-reading instruction architecturally executed if the access generates a Translation fault for any other reason.

`0x82F6`, **DTLB_WALK_ST_RETIRED, Store instruction architecturally executed, at least one translation table walk**

The counter counts each Memory-writing instruction architecturally executed counted by DTLB_WALK_LDST_RETIRED that caused a refill of a data TLB involving at least one translation table walk access.

The counter does not count the Memory-writing instruction architecturally executed if any of the following are true:

- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the Memory-writing instruction architecturally executed if the access generates a Translation fault for any other reason.

`0x82F7`, **DTLB_WALK_LDST_RETIRED, Load or store instruction architecturally executed, at least one translation table walk**

The counter counts each Memory-reading instruction or Memory-writing instruction architecturally executed counted by LDST_ANY_RETIRED that caused a refill of a data TLB involving at least one translation table walk access.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The counter does not count the Memory-reading instruction or Memory-writing instruction architecturally executed if any of the following are true:

- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the Memory-reading instruction or Memory-writing instruction architecturally executed if the access generates a Translation fault for any other reason.

`0x82F8`**, DTLB_WALK_PRF, Data TLB prefetch, with at least one translation table walk**

The counter counts each software prefetch or hardware prefetch access counted by L1D_TLB_PRF that causes a refill or update of a data or unified TLB involving at least one translation table walk access.

The counter does not count the software prefetch or hardware prefetch access if any of the following are true:

- The access is due to a TLB maintenance instruction.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the software prefetch or hardware prefetch access if the access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x82F9`**, ITLB_WALK_PRF, Instruction TLB prefetch, with at least one translation table walk**

The counter counts each software prefetch or hardware prefetch access counted by L1I_TLB_PRF that causes a refill or update of an instruction TLB involving at least one translation table walk access.

The counter does not count the software prefetch or hardware prefetch access if any of the following are true:

- The access is due to a TLB maintenance instruction.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the software prefetch or hardware prefetch access if the access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x82FA`**, DTLB_WALK_HWPRF, Data TLB hardware prefetch, with at least one translation table walk**

The counter counts each hardware prefetch access counted by L1D_TLB_HWPRF that causes a refill or update of a data or unified TLB involving at least one translation table walk access.

The counter does not count the hardware prefetch access if any of the following are true:

- The access is due to a TLB maintenance instruction.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

It is IMPLEMENTATION DEFINED whether the counter counts the hardware prefetch access if the access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x82FB`**, ITLB_WALK_HWPRF, Instruction TLB hardware prefetch, with at least one translation table walk**

The counter counts each hardware prefetch access counted by L1I_TLB_HWPRF that causes a refill or update of an instruction TLB involving at least one translation table walk access.

The counter does not count the hardware prefetch access if any of the following are true:

- The access is due to a TLB maintenance instruction.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the hardware prefetch access if the access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x82FC`**, L1D_TLB_PRF, Level 1 data TLB access, prefetch**

The counter counts each access counted by L1D_TLB that is due to a prefetch instruction, or hardware prefetch.

The counter does not count the access if any of the following are true:

- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the access if the access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x82FD`**, L1I_TLB_PRF, Level 1 instruction TLB access, prefetch**

The counter counts each access counted by L1I_TLB that is due to a prefetch instruction, or hardware prefetch.

The counter does not count the access if any of the following are true:

- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the access if the access generates a Translation fault for any other reason.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x82FE`, **L1D_TLB_HWPRF, Level 1 data TLB access, hardware prefetch**

The counter counts each access counted by L1D_TLB_PRF that is due to a hardware prefetch.

The counter does not count the access if any of the following are true:

- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the access if the access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x82FF`, **L1I_TLB_HWPRF, Level 1 instruction TLB access, hardware prefetch**

The counter counts each access counted by L1I_TLB_PRF that is due to a hardware prefetch.

The counter does not count the access if any of the following are true:

- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the access if the access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x8304`, **DSNP_HITM_LD_RETIRED, Load instruction architecturally executed, snoop hit**

The counter counts each architecturally executed instruction counted by DSNP_HITM_LDST_RETIRED that is a Memory-reading instruction.

`0x8305`, **DSNP_HITM_NEAR_LD_RETIRED, Load instruction architecturally executed, snoop hit in near cache**

The counter counts each architecturally executed instruction counted by DSNP_HITM_NEAR_LDST_RETIRED that is a Memory-reading instruction.

`0x8306`, **DSNP_HITM_FAR_LD_RETIRED, Load instruction architecturally executed, snoop hit in far cache**

The counter counts each architecturally executed instruction counted by DSNP_HITM_FAR_LDST_RETIRED that is a Memory-reading instruction.

`0x8307`, **DSNP_HITM_REMOTE_LD_RETIRED, Load instruction architecturally executed, snoop hit in remote cache**

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The counter counts each architecturally executed instruction counted by DSNP_HITM_REMOTE_LDST_RETIRED that is a Memory-reading instruction.

`0x8308`, **DSNP_HITM_ST_RETIRED, Store instruction architecturally executed, snoop hit**

The counter counts each architecturally executed instruction counted by DSNP_HITM_LDST_RETIRED that is a Memory-writing instruction.

`0x8309`, **DSNP_HITM_NEAR_ST_RETIRED, Store instruction architecturally executed, snoop hit in near cache**

The counter counts each architecturally executed instruction counted by DSNP_HITM_NEAR_LDST_RETIRED that is a Memory-writing instruction.

`0x830A`, **DSNP_HITM_FAR_ST_RETIRED, Store instruction architecturally executed, snoop hit in far cache**

The counter counts each architecturally executed instruction counted by DSNP_HITM_FAR_LDST_RETIRED that is a Memory-writing instruction.

`0x830B`, **DSNP_HITM_REMOTE_ST_RETIRED, Store instruction architecturally executed, snoop hit in remote cache**

The counter counts each architecturally executed instruction counted by DSNP_HITM_REMOTE_LDST_RETIRED that is a Memory-writing instruction.

`0x830C`, **DSNP_HITM_LDST_RETIRED, Load or store instruction architecturally executed, snoop hit**

The counter counts each architecturally executed Memory-reading instruction or Memory-writing instruction counted by LDST_ANY_RETIRED that hit modified data in a data cache outside the cache hierarchy of this PE.

`0x830D`, **DSNP_HITM_NEAR_LDST_RETIRED, Load or store instruction architecturally executed, snoop hit in near cache**

The counter counts each architecturally executed Memory-reading instruction or Memory-writing instruction counted by DSNP_HITM_LDST_RETIRED that hit modified data outside of the cache hierarchy of this PE in the local PE cluster.

`0x830E`, **DSNP_HITM_FAR_LDST_RETIRED, Load or store instruction architecturally executed, snoop hit in far cache**

The counter counts each architecturally executed Memory-reading instruction or Memory-writing instruction counted by DSNP_HITM_LDST_RETIRED that hit modified data outside the local PE cluster on the same device.

`0x830F`, **DSNP_HITM_REMOTE_LDST_RETIRED, Load or store instruction architecturally executed, snoop hit in remote cache**

The counter counts each architecturally executed Memory-reading instruction or Memory-writing instruction counted by DSNP_HITM_LDST_RETIRED that hit modified data on a remote device.

`0x8310`, **ISNP_HIT_RETIRED, Instruction architecturally executed, instruction fetch snoop hit**

The counter counts each instruction architecturally executed counted by INST_RETIRED that hit in an instruction cache outside the cache hierarchy of this PE when fetched.

`0x8311`, **ISNP_HIT_NEAR_RETIRED, Instruction architecturally executed, instruction fetch snoop hit in near cache**

The counter counts each instruction architecturally executed counted by ISNP_HIT_RETIRED that hit outside of the cache hierarchy of this PE in the local PE cluster when fetched.

`0x8312`, **ISNP_HIT_FAR_RETIRED, Instruction architecturally executed, instruction fetch snoop hit in far cache**

The counter counts each instruction architecturally executed counted by ISNP_HIT_RETIRED that hit outside the local PE cluster on the same device when fetched.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x8313`, **ISNP_HIT_REMOTE_RETIRED, Instruction architecturally executed, instruction fetch snoop hit in remote cache**

The counter counts each instruction architecturally executed counted by ISNP_HIT_RETIRED that hit on a remote device when fetched.

`0x8314`, **DSNP_HIT_LD_RETIRED, Load instruction architecturally executed, snoop hit**

The counter counts each architecturally executed instruction counted by DSNP_HIT_LDST_RETIRED that is a Memory-reading instruction.

`0x8315`, **DSNP_HIT_NEAR_LD_RETIRED, Load instruction architecturally executed, snoop hit in near cache**

The counter counts each architecturally executed instruction counted by DSNP_HIT_NEAR_LDST_RETIRED that is a Memory-reading instruction.

`0x8316`, **DSNP_HIT_FAR_LD_RETIRED, Load instruction architecturally executed, snoop hit in far cache**

The counter counts each architecturally executed instruction counted by DSNP_HIT_FAR_LDST_RETIRED that is a Memory-reading instruction.

`0x8317`, **DSNP_HIT_REMOTE_LD_RETIRED, Load instruction architecturally executed, snoop hit in remote cache**

The counter counts each architecturally executed instruction counted by DSNP_HIT_REMOTE_LDST_RETIRED that is a Memory-reading instruction.

`0x8318`, **DSNP_HIT_ST_RETIRED, Store instruction architecturally executed, snoop hit**

The counter counts each architecturally executed instruction counted by DSNP_HIT_LDST_RETIRED that is a Memory-writing instruction.

`0x8319`, **DSNP_HIT_NEAR_ST_RETIRED, Store instruction architecturally executed, snoop hit in near cache**

The counter counts each architecturally executed instruction counted by DSNP_HIT_NEAR_LDST_RETIRED that is a Memory-writing instruction.

`0x831A`, **DSNP_HIT_FAR_ST_RETIRED, Store instruction architecturally executed, snoop hit in far cache**

The counter counts each architecturally executed instruction counted by DSNP_HIT_FAR_LDST_RETIRED that is a Memory-writing instruction.

`0x831B`, **DSNP_HIT_REMOTE_ST_RETIRED, Store instruction architecturally executed, snoop hit in remote cache**

The counter counts each architecturally executed instruction counted by DSNP_HIT_REMOTE_LDST_RETIRED that is a Memory-writing instruction.

`0x831C`, **DSNP_HIT_LDST_RETIRED, Load or store instruction architecturally executed, snoop hit**

The counter counts each architecturally executed Memory-reading instruction or Memory-writing instruction counted by LDST_ANY_RETIRED that hit in a data cache outside the cache hierarchy of this PE.

`0x831D`, **DSNP_HIT_NEAR_LDST_RETIRED, Load or store instruction architecturally executed, snoop hit in near cache**

The counter counts each architecturally executed Memory-reading instruction or Memory-writing instruction counted by DSNP_HIT_LDST_RETIRED that hit outside of the cache hierarchy of this PE in the local PE cluster.

`0x831E`, **DSNP_HIT_FAR_LDST_RETIRED, Load or store instruction architecturally executed, snoop hit in far cache**

The counter counts each architecturally executed Memory-reading instruction or Memory-writing instruction counted by DSNP_HIT_LDST_RETIRED that hit outside the local PE cluster on the same device.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x831F`, **DSNP_HIT_REMOTE_LDST_RETIRED, Load or store instruction architecturally executed, snoop hit in remote cache**

The counter counts each architecturally executed Memory-reading instruction or Memory-writing instruction counted by DSNP_HIT_LDST_RETIRED that hit on a remote device.

`0x8320`, **L1D_CACHE_REFILL_PERCYC, Level 1 data or unified cache refills in progress**

The counter increments by the number of Level 1 data or unified cache refills counted by L1D_CACHE_REFILL in progress on each Processor cycle.

The ratio L1D_CACHE_REFILL_PERCYC ÷ L1D_CACHE_REFILL is the mean duration of Level 1 data or unified cache refills in Processor cycles.

`0x8321`, **L2D_CACHE_REFILL_PERCYC, Level 2 data or unified cache refills in progress**

The counter increments by the number of Level 2 data or unified cache refills counted by L2D_CACHE_REFILL in progress on each Processor cycle.

The ratio L2D_CACHE_REFILL_PERCYC ÷ L2D_CACHE_REFILL is the mean duration of Level 2 data or unified cache refills in Processor cycles.

`0x8322`, **L3D_CACHE_REFILL_PERCYC, Level 3 data or unified cache refills in progress**

The counter increments by the number of Level 3 data or unified cache refills counted by L3D_CACHE_REFILL in progress on each Processor cycle.

The ratio L3D_CACHE_REFILL_PERCYC ÷ L3D_CACHE_REFILL is the mean duration of Level 3 data or unified cache refills in Processor cycles.

`0x8324`, **L1I_CACHE_REFILL_PERCYC, Level 1 instruction or unified cache refills in progress**

The counter increments by the number of Level 1 instruction or unified cache refills counted by L1I_CACHE_REFILL in progress on each Processor cycle.

The ratio L1I_CACHE_REFILL_PERCYC ÷ L1I_CACHE_REFILL is the mean duration of Level 1 instruction or unified cache refills in Processor cycles.

`0x8325`, **L2I_CACHE_REFILL_PERCYC, Level 2 instruction or unified cache refills in progress**

The counter increments by the number of Level 2 instruction or unified cache refills counted by L2I_CACHE_REFILL in progress on each Processor cycle.

The ratio L2I_CACHE_REFILL_PERCYC ÷ L2I_CACHE_REFILL is the mean duration of Level 2 instruction or unified cache refills in Processor cycles.

`0x8328`, **REMOTE_CACHE_HIT_RD, Remote cache demand access hit, read**

The counter counts each cache hit on a remote device due to a demand read.

`0x8329`, **REMOTE_CACHE_HITM_RD, Remote cache demand access hit modified, read**

The counter counts each cache hit on a remote device counted by REMOTE_CACHE_HIT_RD that accesses a previously modified location.

`0x832A`, **REMOTE_LFB_HIT_RD, Remote cache demand access line-fill buffer hit, read**

The counter counts each cache hit on a remote device counted by REMOTE_CACHE_HIT_RD that hits a recently fetched line.

That is, the read hits a cache line that is in the process of being loaded into the cache attached to a remote device, and so does not generate a new refill, but has to wait for the previous refill to complete.

`0x832B`, **REMOTE_CACHE_HIT_PRFM, Remote cache software prefetch hit**

The counter counts each cache hit on a remote device due to a software prefetch.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x8330`, **L1GCS_CACHE, Level 1 GCS cache access**

The counter counts each GCS data read operation or GCS data write operation that causes a cache access to at least a Level 1 cache.

Each access to a cache line is counted including the multiple accesses caused by single instructions such as `GCSPUSHX` or `GCSPOPX`. Each access to other Level 1 memory structures, for example refill buffers, write buffers, and write-back buffers, is also counted.

Accesses to a Level 1 cache due to a GCS hardware prefetcher as well are counted.

When there are separate GCS caching structures implemented:

- It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.
- Accesses that only update the cache status information for a cache entry without accessing the content of the cache entry are not counted.

`0x8331`, **L1GCS_CACHE_RW, Level 1 GCS cache demand access**

The counter counts each access counted by L1GCS_CACHE that is due to a demand Memory-read operation or demand Memory-write operation.

This includes accesses made by Speculatively executed instructions.

Arm recommends that this event is implemented if event L1GCS_CACHE_HWPRF is implemented.

`0x8332`, **L1GCS_CACHE_HWPRF, Level 1 GCS cache hardware prefetch**

The counter counts each access counted by L1GCS_CACHE that is due to a GCS hardware prefetcher.

`0x8334`, **L1GCS_CACHE_MISS, Level 1 GCS cache demand access miss**

The counter counts each access counted by L1GCS_CACHE_RW that misses in the Level 1 caches causing an access to outside of the Level 1 caches of this PE.

`0x8335`, **L1GCS_CACHE_MISS_RD, Level 1 GCS cache demand access read miss**

The counter counts each access counted by L1GCS_CACHE_MISS that is also a Memory-read operation.

It is IMPLEMENTATION DEFINED whether accesses that result from cache maintenance instructions are counted.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

Arm recommends that this event is implemented if event L1GCS_CACHE is implemented.

`0x8336`, **L1GCS_CACHE_HIT_RW, Level 1 GCS cache demand access hit**

The counter counts each access counted by L1GCS_CACHE_RW that hits in a Level 1 cache.

`0x8337`, **L1GCS_CACHE_HIT_RW_FHWPRF, Level 1 GCS cache demand access first hit, fetched by GCS hardware prefetcher**

The counter counts each demand access first hit counted by L1GCS_CACHE_HIT_RW where the cache line was fetched by a GCS hardware prefetcher.

That is, the L1GCS_CACHE_REFILL_HWPRF event was generated when the cache line was fetched into the cache.

Only the first hit by a demand access is counted. After this event is generated for a cache line, the event is not generated again for the same cache line while it remains in the cache.

`0x8338`, **L1GCS_CACHE_REFILL, Level 1 GCS cache refill**

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The counter counts each access counted by L1GCS_CACHE that causes a refill of a Level 1 cache from outside of the Level 1 cache.

A refill includes any access that causes data to be fetched from outside of the Level 1 caches, even if the data is ultimately not allocated into the Level 1 cache. For example, data might be fetched into a buffer but then discarded, rather than being allocated into a cache. These buffers are treated as part of the cache.

For example, the counter counts accesses to the Level 1 cache that cause a refill that is satisfied by fetching data from memory, a Level 2 cache, or a Level 1 cache of another PE.

The counter does not count accesses that:

- Miss in the cache but are satisfied by the refill of a previous miss and do not cause a new refill, even if that previous refill is not complete at the time of the miss.
- Miss in the cache but do not generate a refill, such as a write through the cache.

If the cache is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the cache is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x8339`, **L1GCS_CACHE_REFILL_HWPRF, Level 1 GCS cache refill, hardware prefetch**

The counter counts each hardware prefetch counted by L1GCS_CACHE_HWPRF that causes a refill of a Level 1 cache from outside of the Level 1 cache.

`0x833A`, **L1GCS_CACHE_REFILL_PERCYC, Level 1 GCS cache refills in progress**

The counter increments by the number of Level 1 GCS cache refills counted by L1GCS_CACHE_REFILL in progress on each Processor cycle.

The ratio L1GCS_CACHE_REFILL_PERCYC ÷ L1GCS_CACHE_REFILL is the mean duration of Level 1 GCS cache refills in Processor cycles.

`0x833C`, **L1GCS_LFB_HIT_RW, Level 1 GCS cache demand access line-fill buffer hit**

The counter counts each demand access counted by L1GCS_CACHE_HIT_RW that hits a recently fetched line in a Level 1 cache.

That is, the access hits a cache line that is in the process of being loaded into the Level 1 cache, and so does not generate a new refill, but has to wait for the previous refill to complete.

`0x833D`, **L1GCS_LFB_HIT_RW_FHWPRF, Level 1 GCS cache demand access line-fill buffer first hit, recently fetched by GCS hardware prefetcher**

The counter counts each demand access line-fill buffer first hit counted by L1GCS_LFB_HIT_RW where the cache line was fetched by a GCS hardware prefetcher.

That is, the access hits a cache line that is in the process of being loaded into the Level 1 cache, and so does not generate a new refill, but has to wait for the previous refill to complete, and the L1GCS_CACHE_REFILL_HWPRF event was generated when the cache line was fetched into the cache.

`0x833E`, **L1GCS_CACHE_INVAL, Level 1 GCS cache invalidate**

The counter counts each invalidation of a cache line in a Level 1 GCS cache.

For example:

- Invalidation of a cache line because of a `GCSB` effect.
- Transfer of ownership of a cache line to another cache because of a coherency or refill request.

The counter does not count events if a cache refill of the Level 1 cache invalidates a line in the Level 1 cache.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Software that uses this event must know whether the Level 1 cache is shared with other PEs. This event does not follow the general rule of Level 1 cache events of only counting Attributable events.

`0x8340`**, L1GCS_TLB, Level 1 GCS TLB access**

The counter counts each GCS data read operation or GCS data write operation that causes a TLB access to at least a Level 1 TLB.

Each access to a TLB entry is counted including multiple accesses caused by single instructions such as `GCSPUSHX` or `GCSPOPX`.

The counter does not count the access if the access is due to a TLB maintenance instruction.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x8341`**, L1GCS_TLB_RW, Level 1 GCS demand TLB access**

The counter counts each access counted by L1GCS_TLB that is due to a demand Memory-read operation or demand Memory-write operation.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x8342`**, L1GCS_TLB_HWPRF, Level 1 GCS demand TLB access, GCS hardware prefetch**

The counter counts each access counted by L1GCS_TLB that is due to a GCS hardware prefetch.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x8344`**, GCSTLB_WALK, GCS TLB access with at least one translation table walk**

The counter counts each access counted by L1GCS_TLB that causes a refill of a TLB involving at least one translation table walk access.

This includes each complete or partial translation table walk that causes an access to memory, including to data or translation table walk caches.

Accesses that cause an update of an existing TLB entry involving at least one translation table walk access are counted.

The counter does not count the access if any of the following are true:

- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.
- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- The access is due to a TLB maintenance instruction.

It is IMPLEMENTATION DEFINED whether the counter counts the access if the access generates a Translation fault for any other reason.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The counter does not count the access if any of the following are true:

- The access generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- FEAT_E0PD is implemented and the access is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the access if the access generates a Translation fault for any other reason.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x8345`**, GCSTLB_WALK_RW, GCS TLB demand access with at least one translation table walk**

The counter counts each demand access counted by both GCSTLB_WALK and L1GCS_TLB_RW.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x8346`**, GCSTLB_WALK_PERCYC, Translation table walks in progress**

The counter increments by the number of translation table walks counted by GCSTLB_WALK in progress on each Processor cycle.

The ratio GCSTLB_WALK_PERCYC ÷ GCSTLB_WALK is the mean duration of translation table walks in Processor cycles.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x8347`**, GCSTLB_WALK_STEP, GCS TLB translation table walk, step**

The counter counts each translation table walk access due to a refill of a TLB.

The translation table walk must be due to either GCS data read operation or GCS data write operation.

The event is Attributable to the access that missed in the TLB and caused the walk, not to the owner of the translation tables being accessed. For example, this means that if an EL0 access causes a translation table walk consisting of accesses to both stage 1 and stage 2 translation tables, all accesses are counted if event counting is allowed at EL0, regardless of whether event counting is allowed at EL1 and/or EL2.

The counter does not count the event if any of the following are true:

- The access causing the refill generates a Translation fault because the applicable TCR_ELx.EPDy bit is 1.
- The access causing the refill is an unprivileged access that generates a Translation fault because the applicable TCR_ELx.E0PDy bit is 1.

It is IMPLEMENTATION DEFINED whether the counter counts the event if the access causing the refill generates a Translation fault for any other reason.

Implementation of this optional event requires that FEAT_E0PD is implemented.

If the TLB is shared and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 0, then the counter counts only events Attributable to the PE counting the event. For a multithreaded processor implementation, if the

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TLB is shared by PEs other than the PEs in the multithreaded processor and the Effective value of PMEVTYPER<n>_EL0.MT for the counter is 1, then the counter counts only events Attributable to PEs in the multithreaded processor. In all other cases, it is IMPLEMENTATION DEFINED whether only events Attributable to the PE counting the event or all events are counted, and might depend on the Effective value of PMEVTYPER<n>_EL0.MT.

`0x8350`, **STALL_BACKEND_L1GCS, Backend stall cycles, Level 1 GCS cache**

The counter counts each cycle counted by STALL_BACKEND_MEMBOUND when there is a demand data miss in a first level cache.

The demand data miss must be due to either GCS data read operation or GCS data write operation.

The counter does not count the cycle if any of the following are true:

- The STALL_BACKEND_L2D event is implemented and there is a demand data miss in the second level of data or unified cache, meaning the STALL_BACKEND_L2D event counts the cycle.
- There is a demand data miss in the last level of data or unified cache within the PE clock domain, meaning the STALL_BACKEND_MEM event counts the cycle.

Implementation of this optional event requires that the first level cache is implemented within the PE clock domain and is not the last level cache within the PE clock domain.

Implementation of this optional event requires implementing L1GCS_CACHE event.

Note

It is possible that both the events STALL_BACKEND_L1GCS and STALL_BACKEND_L1D events may be counted on a same cycle.

`0x8351`, **STALL_BACKEND_GCSTLB, Backend stall cycles, GCS TLB**

The counter counts each cycle counted by STALL_BACKEND_MEMBOUND when there is a demand data miss in a TLB.

The demand data miss must be due to translation of a GCS data read operation or GCS data write operation.

Implementation of this optional event requires implementing L1GCS_TLB event.

`0x8353`, **SE_FP_SPEC, Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD floating-point, data-processing**

The counter counts each speculatively executed floating-point data-processing operation counted by FP_SPEC due to an A64 Advanced SIMD, SVE, or SME operation.

That is, each operation counted by ASE_FP_SPEC, SVE_FP_SPEC, or SME_FP_SPEC.

`0x835D`, **SE_SPEC, Operation speculatively executed, Advanced SIMD, SVE, or SME data processing**

The counter counts each operation counted by INST_SPEC that is an Advanced SIMD, SVE, or SME data-processing operation.

See ASE_SVE_SPEC and SME_SPEC for these classifications.

`0x835F`, **SE_INST_SPEC, Operation speculatively executed, Advanced SIMD, SVE, or SME**

The counter counts each speculatively executed operation counted by INST_SPEC that is classified as an Advanced SIMD, SVE, or SME operation.

See ASE_SVE_INST_SPEC and SME_INST_SPEC for these classifications.

`0x8361`, **SE_INT8_SPEC, Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD 8-bit integer, data-processing**

The counter counts each speculatively executed 8-bit integer data-processing operation counted by SE_INT_SPEC due to an A64 Advanced SIMD, SVE, or SME operation.

That is, each operation counted by ASE_INT8_SPEC, SVE_INT8_SPEC, or SME_INT8_SPEC.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x8363`, **SE_FP_BF16_SPEC, Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD BFloat16 floating-point, data-processing**

The counter counts each speculatively executed BFloat16 floating-point data-processing operation counted by SE_FP_SPEC due to an A64 Advanced SIMD, SVE, or SME operation.

That is, each operation counted by SVE_FP_BF16_SPEC, or SME_FP_BF16_SPEC.

`0x8365`, **SE_INT16_SPEC, Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD 16-bit integer, data-processing**

The counter counts each speculatively executed 16-bit integer data-processing operation counted by SE_INT_SPEC due to an A64 Advanced SIMD, SVE, or SME operation.

That is, each operation counted by ASE_INT16_SPEC, SVE_INT16_SPEC, or SME_INT16_SPEC.

`0x8367`, **SE_FP_HP_SPEC, Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD half-precision floating-point, data-processing**

The counter counts each speculatively executed half-precision floating-point data-processing operation counted by SE_FP_SPEC due to an A64 Advanced SIMD, SVE, or SME operation.

That is, each operation counted by ASE_FP_HP_SPEC, SVE_FP_HP_SPEC, or SME_FP_HP_SPEC.

`0x8369`, **SE_INT32_SPEC, Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD 32-bit integer, data-processing**

The counter counts each speculatively executed 32-bit integer data-processing operation counted by SE_INT_SPEC due to an A64 Advanced SIMD, SVE, or SME operation.

That is, each operation counted by ASE_INT32_SPEC, SVE_INT32_SPEC, or SME_INT32_SPEC.

`0x836B`, **SE_FP_SP_SPEC, Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD single-precision floating-point, data-processing**

The counter counts each speculatively executed single-precision floating-point data-processing operation counted by SE_FP_SPEC due to an A64 Advanced SIMD, SVE, or SME operation.

That is, each operation counted by ASE_FP_SP_SPEC, SVE_FP_SP_SPEC, or SME_FP_SP_SPEC.

`0x836D`, **SE_INT64_SPEC, Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD 64-bit integer, data-processing**

The counter counts each speculatively executed 64-bit integer data-processing operation counted by SE_INT_SPEC due to an A64 Advanced SIMD, SVE, or SME operation.

That is, each operation counted by ASE_INT64_SPEC, SVE_INT64_SPEC, or SME_INT64_SPEC.

`0x836F`, **SE_FP_DP_SPEC, Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD double-precision floating-point, data-processing**

The counter counts each speculatively executed double-precision floating-point data-processing operation counted by SE_FP_SPEC due to an A64 Advanced SIMD, SVE, or SME operation.

That is, each operation counted by ASE_FP_DP_SPEC, SVE_FP_DP_SPEC, or SME_FP_DP_SPEC.

`0x8371`, **SE_FP_ADDSUB_SPEC, Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD floating-point addition or subtraction**

The counter counts each speculatively executed floating-point addition or subtraction operation counted by SE_FP_SPEC due to an A64 Advanced SIMD, SVE, or SME operation.

That is, each operation counted by ASE_FP_ADDSUB_SPEC, SVE_FP_ADDSUB_SPEC, or SME_FP_ADDSUB_SPEC.

`0x8373`, **SE_FP_FMA_SPEC, Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD floating-point multiply-add or multiply-subtract**

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The counter counts each speculatively executed floating-point multiply-add, multiply-add long, multiply-subtract, or multiply-subtract long operation counted by SE_FP_SPEC due to an A64 Advanced SIMD, SVE, or SME operation.

That is, each operation counted by ASE_FP_FMA_SPEC, SVE_FP_FMA_SPEC, or SME_FP_FMA_SPEC.

`0x8375`, **SE_FP_DOT_SPEC, Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD floating-point dot product**

The counter counts each speculatively executed floating-point dot product operation counted by SE_FP_SPEC due to any of the following A64 instructions:

That is, each operation counted by ASE_FP_DOT_SPEC, SVE_FP_DOT_SPEC, or SME_FP_DOT_SPEC.

`0x8379`, **SE_INT_SPEC, Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD integer, data-processing**

The counter counts each speculatively executed integer data-processing operation counted by INT_SPEC due to an A64 Advanced SIMD, SVE, or SME operation.

That is, each operation counted by ASE_INT_SPEC, SVE_INT_SPEC, or SME_INT_SPEC.

`0x837B`, **SE_INT_MUL_SPEC, Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD integer multiply or multiply-accumulate**

The counter counts each speculatively executed integer multiply, multiply-add, or multiply-subtract operation counted by SE_INT_SPEC due to an A64 Advanced SIMD, SVE, or SME operation.

That is, each operation counted by ASE_INT_MUL_SPEC, SVE_INT_MUL_SPEC, or SME_INT_MUL_SPEC.

`0x837D`, **SE_INT_DOT_SPEC, Operation speculatively executed, Advanced SIMD, SVE, or SME SIMD integer dot product**

The counter counts each speculatively executed integer dot product operation counted by SE_INT_SPEC due to any of the following A64 instructions:

- When FEAT_DotProd is implemented, Advanced SIMD: SDOT or UDOT.

That is, each operation counted by ASE_INT_DOT_SPEC, SVE_INT_DOT_SPEC, or SME_INT_DOT_SPEC.

`0x8380`, **ZA_ACTIVE, CPU cycles, ZA enabled**

The counter counts each cycle counted by CPU_CYCLES when PSTATE.ZA is 1.

`0x8398`, **ASE_LUT_SPEC, Lookup table operation speculatively executed, Advanced SIMD data processing**

The counter counts each speculatively executed LUT operation counted by SE_LUT_SPEC that returns a value from a lookup table made up of a V register which is indexed by values from a V register.

`0x8399`, **SVE_LUT_SPEC, Lookup table operation speculatively executed, SVE data processing**

The counter counts each speculatively executed LUT operation counted by SE_LUT_SPEC that returns a value from a lookup table made up of a Z register which is indexed by values from a Z register.

That is, due to any of the following instructions:

`0x839B`, **SE_LUT_SPEC, Lookup table operation speculatively executed, Advanced SIMD, SVE, or SME data processing**

The counter counts each speculatively executed LUT operation counted by SE_INST_SPEC that returns a value from a lookup table made up of a ZT register, Z or V register which is indexed by values from a Z or V register.

`0x83A0`, **N1_CACHE_HIT_RD, Cache at distance 1, demand access hit, read**

The counter counts each cache hit at distance 1 due to a demand read.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x83A1`**, N2_CACHE_HIT_RD, Cache at distance 2, demand access hit, read**

The counter counts each cache hit at distance 2 due to a demand read.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

Cache distance 2 represents a further distance of travel than cache distance 1.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x83A2`**, N3_CACHE_HIT_RD, Cache at distance 3, demand access hit, read**

The counter counts each cache hit at distance 3 due to a demand read.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

Cache distance 3 represents a further distance of travel than cache distance 2.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x83A3`**, N4_CACHE_HIT_RD, Cache at distance 4, demand access hit, read**

The counter counts each cache hit at distance 4 due to a demand read.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

Cache distance 4 represents a further distance of travel than cache distance 3.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x83A4`**, N1_CACHE_HIT_PRFM, Cache at distance 1, software prefetch hit**

The counter counts each cache hit at distance 1 due to a software prefetch.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x83A5`**, N2_CACHE_HIT_PRFM, Cache at distance 2, software prefetch hit**

The counter counts each cache hit at distance 2 due to a software prefetch.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

Cache distance 2 represents a further distance of travel than cache distance 1.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x83A6`, **N3_CACHE_HIT_PRFM, Cache at distance 3, software prefetch hit**

The counter counts each cache hit at distance 3 due to a software prefetch.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

Cache distance 3 represents a further distance of travel than cache distance 2.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x83A7`, **N4_CACHE_HIT_PRFM, Cache at distance 4, software prefetch hit**

The counter counts each cache hit at distance 4 due to a software prefetch.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

Cache distance 4 represents a further distance of travel than cache distance 3.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x83A8`, **N1_CACHE1_HIT_RD, Cache type 1 at distance 1 demand access hit, read**

The counter counts each demand read counted by N1_CACHE_HIT_RD due to a demand read accessing a first cache type at distance 1.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83A9`, **N2_CACHE1_HIT_RD, Cache type 1 at distance 2 demand access hit, read**

The counter counts each demand read counted by N2_CACHE_HIT_RD due to a demand read accessing a first cache type at distance 2.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83AA`, **N3_CACHE1_HIT_RD, Cache type 1 at distance 3 demand access hit, read**

The counter counts each demand read counted by N3_CACHE_HIT_RD due to a demand read accessing a first cache type at distance 3.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83AB`, **N4_CACHE1_HIT_RD, Cache type 1 at distance 4 demand access hit, read**

The counter counts each demand read counted by N4_CACHE_HIT_RD due to a demand read accessing a first cache type at distance 4.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83AC`, **N1_CACHE1_HIT_PRFM, Cache type 1 at distance 1 software prefetch hit**

The counter counts each software prefetch counted by N1_CACHE_HIT_PRFM due to a software prefetch accessing a first cache type at distance 1.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83AD`, **N2_CACHE1_HIT_PRFM, Cache type 1 at distance 2 software prefetch hit**

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The counter counts each software prefetch counted by N2_CACHE_HIT_PRFM due to a software prefetch accessing a first cache type at distance 2.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83AE`, **N3_CACHE1_HIT_PRFM, Cache type 1 at distance 3 software prefetch hit**

The counter counts each software prefetch counted by N3_CACHE_HIT_PRFM due to a software prefetch accessing a first cache type at distance 3.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83AF`, **N4_CACHE1_HIT_PRFM, Cache type 1 at distance 4 software prefetch hit**

The counter counts each software prefetch counted by N4_CACHE_HIT_PRFM due to a software prefetch accessing a first cache type at distance 4.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83B0`, **N1_CACHE2_HIT_RD, Cache type 2 at distance 1 demand access hit, read**

The counter counts each demand read counted by N1_CACHE_HIT_RD due to a demand read accessing a second cache type at distance 1.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83B1`, **N2_CACHE2_HIT_RD, Cache type 2 at distance 2 demand access hit, read**

The counter counts each demand read counted by N2_CACHE_HIT_RD due to a demand read accessing a second cache type at distance 2.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83B2`, **N3_CACHE2_HIT_RD, Cache type 2 at distance 3 demand access hit, read**

The counter counts each demand read counted by N3_CACHE_HIT_RD due to a demand read accessing a second cache type at distance 3.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83B3`, **N4_CACHE2_HIT_RD, Cache type 2 at distance 4 demand access hit, read**

The counter counts each demand read counted by N4_CACHE_HIT_RD due to a demand read accessing a second cache type at distance 4.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83B4`, **N1_CACHE2_HIT_PRFM, Cache type 2 at distance 1 software prefetch hit**

The counter counts each software prefetch counted by N1_CACHE_HIT_PRFM due to a software prefetch accessing a second cache type at distance 1.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83B5`, **N2_CACHE2_HIT_PRFM, Cache type 2 at distance 2 software prefetch hit**

The counter counts each software prefetch counted by N2_CACHE_HIT_PRFM due to a software prefetch accessing a second cache type at distance 2.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x83B6`, **N3_CACHE2_HIT_PRFM, Cache type 2 at distance 3 software prefetch hit**

The counter counts each software prefetch counted by N3_CACHE_HIT_PRFM due to a software prefetch accessing a second cache type at distance 3.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83B7`, **N4_CACHE2_HIT_PRFM, Cache type 2 at distance 4 software prefetch hit**

The counter counts each software prefetch counted by N4_CACHE_HIT_PRFM due to a software prefetch accessing a second cache type at distance 4.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83B8`, **N1_CACHE3_HIT_RD, Cache type 3 at distance 1 demand access hit, read**

The counter counts each demand read counted by N1_CACHE_HIT_RD due to a demand read accessing a third cache type at distance 1.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83B9`, **N2_CACHE3_HIT_RD, Cache type 3 at distance 2 demand access hit, read**

The counter counts each demand read counted by N2_CACHE_HIT_RD due to a demand read accessing a third cache type at distance 2.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83BA`, **N3_CACHE3_HIT_RD, Cache type 3 at distance 3 demand access hit, read**

The counter counts each demand read counted by N3_CACHE_HIT_RD due to a demand read accessing a third cache type at distance 3.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83BB`, **N4_CACHE3_HIT_RD, Cache type 3 at distance 4 demand access hit, read**

The counter counts each demand read counted by N4_CACHE_HIT_RD due to a demand read accessing a third cache type at distance 4.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83BC`, **N1_CACHE3_HIT_PRFM, Cache type 3 at distance 1 software prefetch hit**

The counter counts each software prefetch counted by N1_CACHE_HIT_PRFM due to a software prefetch accessing a third cache type at distance 1.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83BD`, **N2_CACHE3_HIT_PRFM, Cache type 3 at distance 2 software prefetch hit**

The counter counts each software prefetch counted by N2_CACHE_HIT_PRFM due to a software prefetch accessing a third cache type at distance 2.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83BE`, **N3_CACHE3_HIT_PRFM, Cache type 3 at distance 3 software prefetch hit**

The counter counts each software prefetch counted by N3_CACHE_HIT_PRFM due to a software prefetch accessing a third cache type at distance 3.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83BF`, **N4_CACHE3_HIT_PRFM, Cache type 3 at distance 4 software prefetch hit**

The counter counts each software prefetch counted by N4_CACHE_HIT_PRFM due to a software prefetch accessing a third cache type at distance 4.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83C0`, **N1_CACHE_HITM_RD, Cache at distance 1, demand access hit modified, read**

The counter counts each demand read counted by N1_CACHE_HIT_RD that accesses a previously modified location in a cache at distance 1.

`0x83C1`, **N2_CACHE_HITM_RD, Cache at distance 2, demand access hit modified, read**

The counter counts each demand read counted by N2_CACHE_HIT_RD that accesses a previously modified location in a cache at distance 2.

`0x83C2`, **N3_CACHE_HITM_RD, Cache at distance 3, demand access hit modified, read**

The counter counts each demand read counted by N3_CACHE_HIT_RD that accesses a previously modified location in a cache at distance 3.

`0x83C3`, **N4_CACHE_HITM_RD, Cache at distance 4, demand access hit modified, read**

The counter counts each demand read counted by N4_CACHE_HIT_RD that accesses a previously modified location in a cache at distance 4.

`0x83C4`, **N1_LFB_HIT_RD, Cache at distance 1 demand access line-fill buffer hit, read**

The counter counts each demand read counted by N1_CACHE_HIT_RD that hits a recently fetched line in a cache at distance 1.

That is, the read hits a cache line that is in the process of being loaded into the cache at distance 1, and so does not generate a new refill, but has to wait for the previous refill to complete.

`0x83C5`, **N2_LFB_HIT_RD, Cache at distance 2 demand access line-fill buffer hit, read**

The counter counts each demand read counted by N2_CACHE_HIT_RD that hits a recently fetched line in a cache at distance 2.

That is, the read hits a cache line that is in the process of being loaded into the cache at distance 2, and so does not generate a new refill, but has to wait for the previous refill to complete.

`0x83C6`, **N3_LFB_HIT_RD, Cache at distance 3 demand access line-fill buffer hit, read**

The counter counts each demand read counted by N3_CACHE_HIT_RD that hits a recently fetched line in a cache at distance 3.

That is, the read hits a cache line that is in the process of being loaded into the cache at distance 3, and so does not generate a new refill, but has to wait for the previous refill to complete.

`0x83C7`, **N4_LFB_HIT_RD, Cache at distance 4 demand access line-fill buffer hit, read**

The counter counts each demand read counted by N4_CACHE_HIT_RD that hits a recently fetched line in a cache at distance 4.

That is, the read hits a cache line that is in the process of being loaded into the cache at distance 4, and so does not generate a new refill, but has to wait for the previous refill to complete.

`0x83C8`, **N1_CACHE1_HITM_RD, Cache type 1 at distance 1 demand access hit modified, read**

The counter counts each demand read counted by N1_CACHE_HITM_RD that accesses a previously modified location in a first cache type at distance 1.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83C9`, **N2_CACHE1_HITM_RD, Cache type 1 at distance 2 demand access hit modified, read**

The counter counts each demand read counted by N2_CACHE_HITM_RD that accesses a previously modified location in a first cache type at distance 2.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83CA`, **N3_CACHE1_HITM_RD, Cache type 1 at distance 3 demand access hit modified, read**

The counter counts each demand read counted by N3_CACHE_HITM_RD that accesses a previously modified location in a first cache type at distance 3.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83CB`, **N4_CACHE1_HITM_RD, Cache type 1 at distance 4 demand access hit modified, read**

The counter counts each demand read counted by N4_CACHE_HITM_RD that accesses a previously modified location in a first cache type at distance 4.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83CC`, **N1_LFB1_HIT_RD, Cache type 1 at distance 1 demand access line-fill buffer hit, read**

The counter counts each demand read counted by N1_LFB_HIT_RD that hits a recently fetched line in a first cache type at distance 1.

That is, the read hits a cache line that is in the process of being loaded into the cache at distance 1, and so does not generate a new refill, but has to wait for the previous refill to complete.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83CD`, **N2_LFB1_HIT_RD, Cache type 1 at distance 2 demand access line-fill buffer hit, read**

The counter counts each demand read counted by N2_LFB_HIT_RD that hits a recently fetched line in a first cache type at distance 2.

That is, the read hits a cache line that is in the process of being loaded into the cache at distance 2, and so does not generate a new refill, but has to wait for the previous refill to complete.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83CE`, **N3_LFB1_HIT_RD, Cache type 1 at distance 3 demand access line-fill buffer hit, read**

The counter counts each demand read counted by N3_LFB_HIT_RD that hits a recently fetched line in a first cache type at distance 3.

That is, the read hits a cache line that is in the process of being loaded into the cache at distance 3, and so does not generate a new refill, but has to wait for the previous refill to complete.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83CF`, **N4_LFB1_HIT_RD, Cache type 1 at distance 4 demand access line-fill buffer hit, read**

The counter counts each demand read counted by N4_LFB_HIT_RD that hits a recently fetched line in a first cache type at distance 4.

That is, the read hits a cache line that is in the process of being loaded into the cache at distance 4, and so does not generate a new refill, but has to wait for the previous refill to complete.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83D0`, **N1_CACHE2_HITM_RD, Cache type 2 at distance 1 demand access hit modified, read**

The counter counts each demand read counted by N1_CACHE_HITM_RD that accesses a previously modified location in a second cache type at distance 1.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83D1`, **N2_CACHE2_HITM_RD, Cache type 2 at distance 2 demand access hit modified, read**

The counter counts each demand read counted by N2_CACHE_HITM_RD that accesses a previously modified location in a second cache type at distance 2.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83D2`, **N3_CACHE2_HITM_RD, Cache type 2 at distance 3 demand access hit modified, read**

The counter counts each demand read counted by N3_CACHE_HITM_RD that accesses a previously modified location in a second cache type at distance 3.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83D3`, **N4_CACHE2_HITM_RD, Cache type 2 at distance 4 demand access hit modified, read**

The counter counts each demand read counted by N4_CACHE_HITM_RD that accesses a previously modified location in a second cache type at distance 4.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83D4`, **N1_LFB2_HIT_RD, Cache type 2 at distance 1 demand access line-fill buffer hit, read**

The counter counts each demand read counted by N1_LFB_HIT_RD that hits a recently fetched line in a second cache type at distance 1.

That is, the read hits a cache line that is in the process of being loaded into the cache at distance 1, and so does not generate a new refill, but has to wait for the previous refill to complete.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83D5`, **N2_LFB2_HIT_RD, Cache type 2 at distance 2 demand access line-fill buffer hit, read**

The counter counts each demand read counted by N2_LFB_HIT_RD that hits a recently fetched line in a second cache type at distance 2.

That is, the read hits a cache line that is in the process of being loaded into the cache at distance 2, and so does not generate a new refill, but has to wait for the previous refill to complete.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83D6`, **N3_LFB2_HIT_RD, Cache type 2 at distance 3 demand access line-fill buffer hit, read**

The counter counts each demand read counted by N3_LFB_HIT_RD that hits a recently fetched line in a second cache type at distance 3.

That is, the read hits a cache line that is in the process of being loaded into the cache at distance 3, and so does not generate a new refill, but has to wait for the previous refill to complete.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x83D7`, **N4_LFB2_HIT_RD, Cache type 2 at distance 4 demand access line-fill buffer hit, read**

The counter counts each demand read counted by N4_LFB_HIT_RD that hits a recently fetched line in a second cache type at distance 4.

That is, the read hits a cache line that is in the process of being loaded into the cache at distance 4, and so does not generate a new refill, but has to wait for the previous refill to complete.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83D8`, **N1_CACHE3_HITM_RD, Cache type 3 at distance 1 demand access hit modified, read**

The counter counts each demand read counted by N1_CACHE_HITM_RD that accesses a previously modified location in a third cache type at distance 1.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83D9`, **N2_CACHE3_HITM_RD, Cache type 3 at distance 2 demand access hit modified, read**

The counter counts each demand read counted by N2_CACHE_HITM_RD that accesses a previously modified location in a third cache type at distance 2.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83DA`, **N3_CACHE3_HITM_RD, Cache type 3 at distance 3 demand access hit modified, read**

The counter counts each demand read counted by N3_CACHE_HITM_RD that accesses a previously modified location in a third cache type at distance 3.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83DB`, **N4_CACHE3_HITM_RD, Cache type 3 at distance 4 demand access hit modified, read**

The counter counts each demand read counted by N4_CACHE_HITM_RD that accesses a previously modified location in a third cache type at distance 4.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83DC`, **N1_LFB3_HIT_RD, Cache type 3 at distance 1 demand access line-fill buffer hit, read**

The counter counts each demand read counted by N1_LFB_HIT_RD that hits a recently fetched line in a third cache type at distance 1.

That is, the read hits a cache line that is in the process of being loaded into the cache at distance 1, and so does not generate a new refill, but has to wait for the previous refill to complete.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83DD`, **N2_LFB3_HIT_RD, Cache type 3 at distance 2 demand access line-fill buffer hit, read**

The counter counts each demand read counted by N2_LFB_HIT_RD that hits a recently fetched line in a third cache type at distance 2.

That is, the read hits a cache line that is in the process of being loaded into the cache at distance 2, and so does not generate a new refill, but has to wait for the previous refill to complete.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83DE`, **N3_LFB3_HIT_RD, Cache type 3 at distance 3 demand access line-fill buffer hit, read**

The counter counts each demand read counted by N3_LFB_HIT_RD that hits a recently fetched line in a third cache type at distance 3.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

That is, the read hits a cache line that is in the process of being loaded into the cache at distance 3, and so does not generate a new refill, but has to wait for the previous refill to complete.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83DF`, **N4_LFB3_HIT_RD, Cache type 3 at distance 4 demand access line-fill buffer hit, read**

The counter counts each demand read counted by N4_LFB_HIT_RD that hits a recently fetched line in a third cache type at distance 4.

That is, the read hits a cache line that is in the process of being loaded into the cache at distance 4, and so does not generate a new refill, but has to wait for the previous refill to complete.

The definition of cache types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83E0`, **N1_MEM_RD, Access to memory at distance 1, demand access, read**

The counter counts each demand read to memory at distance 1.

The memory distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x83E1`, **N2_MEM_RD, Access to memory at distance 2, demand access, read**

The counter counts each demand read to memory at distance 2.

The memory distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

Distance 2 represents a further distance of travel than distance 1.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x83E2`, **N3_MEM_RD, Access to memory at distance 3, demand access, read**

The counter counts each demand read to memory at distance 3.

The memory distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

Distance 3 represents a further distance of travel than distance 2.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x83E3`, **N4_MEM_RD, Access to memory at distance 4, demand access, read**

The counter counts each demand read to memory at distance 4.

The memory distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

Distance 4 represents a further distance of travel than distance 3.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x83E4`, **N1_MEM_PRFM, Access to memory at distance 1, software prefetch**

The counter counts each software prefetch to memory at distance 1.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The memory distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x83E5`**, N2_MEM_PRFM, Access to memory at distance 2, software prefetch**

The counter counts each software prefetch to memory at distance 2.

The memory distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

Distance 2 represents a further distance of travel than distance 1.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x83E6`**, N3_MEM_PRFM, Access to memory at distance 3, software prefetch**

The counter counts each software prefetch to memory at distance 3.

The memory distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

Distance 3 represents a further distance of travel than distance 2.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x83E7`**, N4_MEM_PRFM, Access to memory at distance 4, software prefetch**

The counter counts each software prefetch to memory at distance 4.

The memory distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

Distance 4 represents a further distance of travel than distance 3.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x83E8`**, N1_MEM1_RD, Access to type 1 memory at distance 1, demand access, read**

The counter counts each demand read counted by N1_MEM_RD to a first memory type at distance 1.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system. See also N1_MEM1_RD.

For example, the first memory type might be commodity DRAM connected to a device. However, this must be the same for all devices in the system, such that, for example, N1_MEM1_RD refers to commodity DRAM for a PE at distance 1 from the memory, regardless of which device the PE is located on, and N4_MEM1_RD refers to the same memory for any PE located at distance 4. This means that when a process counting the N1_MEM1_RD event migrates between PEs on different devices, its relative view of the first memory type remains unchanged, even if the distance changes.

`0x83E9`**, N2_MEM1_RD, Access to type 1 memory at distance 2, demand access, read**

The counter counts each demand read counted by N2_MEM_RD to a first memory type at distance 2.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system. See also N1_MEM1_RD.

`0x83EA`**, N3_MEM1_RD, Access to type 1 memory at distance 3, demand access, read**

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The counter counts each demand read counted by N3_MEM_RD to a first memory type at distance 3.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system. See also N1_MEM1_RD.

`0x83EB`, **N4_MEM1_RD, Access to type 1 memory at distance 4, demand access, read**

The counter counts each demand read counted by N4_MEM_RD to a first memory type at distance 4.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system. See also N1_MEM1_RD.

`0x83EC`, **N1_MEM1_PRFM, Access to type 1 memory at distance 1, software prefetch**

The counter counts each software prefetch counted by N1_MEM_PRFM to a first memory type at distance 1.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83ED`, **N2_MEM1_PRFM, Access to type 1 memory at distance 2, software prefetch**

The counter counts each software prefetch counted by N2_MEM_PRFM to a first memory type at distance 2.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83EE`, **N3_MEM1_PRFM, Access to type 1 memory at distance 3, software prefetch**

The counter counts each software prefetch counted by N3_MEM_PRFM to a first memory type at distance 3.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83EF`, **N4_MEM1_PRFM, Access to type 1 memory at distance 4, software prefetch**

The counter counts each software prefetch counted by N4_MEM_PRFM to a first memory type at distance 4.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83F0`, **N1_MEM2_RD, Access to type 2 memory at distance 1, demand access, read**

The counter counts each demand read counted by N1_MEM_RD to a second memory type at distance 1.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system. See also N1_MEM2_RD.

For example, the second memory type might be bulk expansion memory connected to a device. However, this must be the same for all devices in the system, such that, for example, N1_MEM2_RD refers to bulk expansion memory for a PE at distance 1 from the memory, regardless of which device the PE is located on, and N4_MEM2_RD refers to the same memory for any PE located at distance 4. This means that when a process counting the N1_MEM2_RD event migrates between PEs on different devices, its relative view of the second memory type remains unchanged, even if the distance changes.

`0x83F1`, **N2_MEM2_RD, Access to type 2 memory at distance 2, demand access, read**

The counter counts each demand read counted by N2_MEM_RD to a second memory type at distance 2.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system. See also N1_MEM2_RD.

`0x83F2`, **N3_MEM2_RD, Access to type 2 memory at distance 3, demand access, read**

The counter counts each demand read counted by N3_MEM_RD to a second memory type at distance 3.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system. See also N1_MEM2_RD.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x83F3`, **N4_MEM2_RD, Access to type 2 memory at distance 4, demand access, read**

The counter counts each demand read counted by N4_MEM_RD to a second memory type at distance 4.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system. See also N1_MEM2_RD.

`0x83F4`, **N1_MEM2_PRFM, Access to type 2 memory at distance 1, software prefetch**

The counter counts each software prefetch counted by N1_MEM_PRFM to a second memory type at distance 1.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83F5`, **N2_MEM2_PRFM, Access to type 2 memory at distance 2, software prefetch**

The counter counts each software prefetch counted by N2_MEM_PRFM to a second memory type at distance 2.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83F6`, **N3_MEM2_PRFM, Access to type 2 memory at distance 3, software prefetch**

The counter counts each software prefetch counted by N3_MEM_PRFM to a second memory type at distance 3.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83F7`, **N4_MEM2_PRFM, Access to type 2 memory at distance 4, software prefetch**

The counter counts each software prefetch counted by N4_MEM_PRFM to a second memory type at distance 4.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83F8`, **N1_MEM3_RD, Access to type 3 memory at distance 1, demand access, read**

The counter counts each demand read counted by N1_MEM_RD to a third memory type at distance 1.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system. See also N1_MEM3_RD.

For example, the third memory type might be high-bandwidth memory connected to a device. However, this must be the same for all devices in the system, such that, for example, N1_MEM3_RD refers to high-bandwidth memory for a PE at distance 1 from the memory, regardless of which device the PE is located on, and N4_MEM3_RD refers to the same memory for any PE located at distance 4. This means that when a process counting the N1_MEM3_RD event migrates between PEs on different devices, its relative view of the third memory type remains unchanged, even if the distance changes.

`0x83F9`, **N2_MEM3_RD, Access to type 3 memory at distance 2, demand access, read**

The counter counts each demand read counted by N2_MEM_RD to a third memory type at distance 2.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system. See also N1_MEM3_RD.

`0x83FA`, **N3_MEM3_RD, Access to type 3 memory at distance 3, demand access, read**

The counter counts each demand read counted by N3_MEM_RD to a third memory type at distance 3.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system. See also N1_MEM3_RD.

`0x83FB`, **N4_MEM3_RD, Access to type 3 memory at distance 4, demand access, read**

The counter counts each demand read counted by N4_MEM_RD to a third memory type at distance 4.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system. See also N1_MEM3_RD.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x83FC`, **N1_MEM3_PRFM, Access to type 3 memory at distance 1, software prefetch**

The counter counts each software prefetch counted by N1_MEM_PRFM to a third memory type at distance 1.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83FD`, **N2_MEM3_PRFM, Access to type 3 memory at distance 2, software prefetch**

The counter counts each software prefetch counted by N2_MEM_PRFM to a third memory type at distance 2.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83FE`, **N3_MEM3_PRFM, Access to type 3 memory at distance 3, software prefetch**

The counter counts each software prefetch counted by N3_MEM_PRFM to a third memory type at distance 3.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x83FF`, **N4_MEM3_PRFM, Access to type 3 memory at distance 4, software prefetch**

The counter counts each software prefetch counted by N4_MEM_PRFM to a third memory type at distance 4.

The definition of memory types in the system is IMPLEMENTATION DEFINED, but must be similar for all devices in the system.

`0x8400`, **ISNP_HIT_N1_RD, Snoop hit in cache at distance 1, demand instruction fetch**

The counter counts each snoop generated by the PE in response to a demand Instruction memory access counted by ISNP_HIT_RD that hits in a cache at distance 1.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x8401`, **ISNP_HIT_N2_RD, Snoop hit in cache at distance 2, demand instruction fetch**

The counter counts each snoop generated by the PE in response to a demand Instruction memory access counted by ISNP_HIT_RD that hits in a cache at distance 2.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x8402`, **ISNP_HIT_N3_RD, Snoop hit in cache at distance 3, demand instruction fetch**

The counter counts each snoop generated by the PE in response to a demand Instruction memory access counted by ISNP_HIT_RD that hits in a cache at distance 3.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x8403`, **ISNP_HIT_N4_RD, Snoop hit in cache at distance 4, demand instruction fetch**

The counter counts each snoop generated by the PE in response to a demand Instruction memory access counted by ISNP_HIT_RD that hits in a cache at distance 4.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x8404`, **DSNP_HIT_N1_RD, Snoop hit in cache at distance 1, demand data read**

The counter counts each snoop generated by the PE in response to a demand Memory-read operation counted by DSNP_HIT_RD that hits in a cache at distance 1.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x8405`, **DSNP_HIT_N2_RD, Snoop hit in cache at distance 2, demand data read**

The counter counts each snoop generated by the PE in response to a demand Memory-read operation counted by DSNP_HIT_RD that hits in a cache at distance 2.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x8406`, **DSNP_HIT_N3_RD, Snoop hit in cache at distance 3, demand data read**

The counter counts each snoop generated by the PE in response to a demand Memory-read operation counted by DSNP_HIT_RD that hits in a cache at distance 3.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x8407`, **DSNP_HIT_N4_RD, Snoop hit in cache at distance 4, demand data read**

The counter counts each snoop generated by the PE in response to a demand Memory-read operation counted by DSNP_HIT_RD that hits in a cache at distance 4.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x8408`, **DSNP_HIT_N1_WR, Snoop hit in cache at distance 1, demand data write**

The counter counts each snoop generated by the PE in response to a demand Memory-write operation counted by DSNP_HIT_WR that hits in a cache at distance 1.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x8409`, **DSNP_HIT_N2_WR, Snoop hit in cache at distance 2, demand data write**

The counter counts each snoop generated by the PE in response to a demand Memory-write operation counted by DSNP_HIT_WR that hits in a cache at distance 2.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x840A`, **DSNP_HIT_N3_WR, Snoop hit in cache at distance 3, demand data write**

The counter counts each snoop generated by the PE in response to a demand Memory-write operation counted by DSNP_HIT_WR that hits in a cache at distance 3.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x840B`, **DSNP_HIT_N4_WR, Snoop hit in cache at distance 4, demand data write**

The counter counts each snoop generated by the PE in response to a demand Memory-write operation counted by DSNP_HIT_WR that hits in a cache at distance 4.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x840C`, **DSNP_HIT_N1_RW, Snoop hit in cache at distance 1, demand data access**

The counter counts each snoop generated by the PE in response to a demand Memory-read operation or demand Memory-write operation counted by DSNP_HIT_RW that hits in a cache at distance 1.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x840D`, **DSNP_HIT_N2_RW, Snoop hit in cache at distance 2, demand data access**

The counter counts each snoop generated by the PE in response to a demand Memory-read operation or demand Memory-write operation counted by DSNP_HIT_RW that hits in a cache at distance 2.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x840E`, **DSNP_HIT_N3_RW, Snoop hit in cache at distance 3, demand data access**

The counter counts each snoop generated by the PE in response to a demand Memory-read operation or demand Memory-write operation counted by DSNP_HIT_RW that hits in a cache at distance 3.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x840F`, **DSNP_HIT_N4_RW, Snoop hit in cache at distance 4, demand data access**

The counter counts each snoop generated by the PE in response to a demand Memory-read operation or demand Memory-write operation counted by DSNP_HIT_RW that hits in a cache at distance 4.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x8410`, **DSNP_HIT_N1_PRFM, Snoop hit in cache at distance 1, software data prefetch**

The counter counts each snoop generated by the PE in response to a data prefetch instruction counted by DSNP_HIT_PRFM that hits in a cache at distance 1.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x8411`, **DSNP_HIT_N2_PRFM, Snoop hit in cache at distance 2, software data prefetch**

The counter counts each snoop generated by the PE in response to a data prefetch instruction counted by DSNP_HIT_PRFM that hits in a cache at distance 2.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x8412`, **DSNP_HIT_N3_PRFM, Snoop hit in cache at distance 3, software data prefetch**

The counter counts each snoop generated by the PE in response to a data prefetch instruction counted by DSNP_HIT_PRFM that hits in a cache at distance 3.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x8413`, **DSNP_HIT_N4_PRFM, Snoop hit in cache at distance 4, software data prefetch**

The counter counts each snoop generated by the PE in response to a data prefetch instruction counted by DSNP_HIT_PRFM that hits in a cache at distance 4.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x8414`, **DSNP_HIT_N1_HWPRF, Snoop hit in cache at distance 1, hardware data prefetch**

The counter counts each snoop generated by a data hardware prefetch counted by DSNP_HIT_HWPRF that hits in a cache at distance 1.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x8415`, **DSNP_HIT_N2_HWPRF, Snoop hit in cache at distance 2, hardware data prefetch**

The counter counts each snoop generated by a data hardware prefetch counted by DSNP_HIT_HWPRF that hits in a cache at distance 2.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x8416`, **DSNP_HIT_N3_HWPRF, Snoop hit in cache at distance 3, hardware data prefetch**

The counter counts each snoop generated by a data hardware prefetch counted by DSNP_HIT_HWPRF that hits in a cache at distance 3.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x8417`**, DSNP_HIT_N4_HWPRF, Snoop hit in cache at distance 4, hardware data prefetch**

The counter counts each snoop generated by a data hardware prefetch counted by DSNP_HIT_HWPRF that hits in a cache at distance 4.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x8418`**, ISNP_HIT_N1_PRFM, Snoop hit in cache at distance 1, software instruction prefetch**

The counter counts each snoop generated by the PE in response to an instruction prefetch instruction counted by ISNP_HIT_PRFM that hits in a cache at distance 1.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x8419`**, ISNP_HIT_N2_PRFM, Snoop hit in cache at distance 2, software instruction prefetch**

The counter counts each snoop generated by the PE in response to an instruction prefetch instruction counted by ISNP_HIT_PRFM that hits in a cache at distance 2.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x841A`**, ISNP_HIT_N3_PRFM, Snoop hit in cache at distance 3, software instruction prefetch**

The counter counts each snoop generated by the PE in response to an instruction prefetch instruction counted by ISNP_HIT_PRFM that hits in a cache at distance 3.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x841B`**, ISNP_HIT_N4_PRFM, Snoop hit in cache at distance 4, software instruction prefetch**

The counter counts each snoop generated by the PE in response to an instruction prefetch instruction counted by ISNP_HIT_PRFM that hits in a cache at distance 4.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x841C`, **ISNP_HIT_N1_HWPRF, Snoop hit in cache at distance 1, hardware instruction prefetch**

The counter counts each snoop generated by an instruction hardware prefetch counted by ISNP_HIT_HWPRF that hits in a cache at distance 1.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x841D`, **ISNP_HIT_N2_HWPRF, Snoop hit in cache at distance 2, hardware instruction prefetch**

The counter counts each snoop generated by an instruction hardware prefetch counted by ISNP_HIT_HWPRF that hits in a cache at distance 2.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x841E`, **ISNP_HIT_N3_HWPRF, Snoop hit in cache at distance 3, hardware instruction prefetch**

The counter counts each snoop generated by an instruction hardware prefetch counted by ISNP_HIT_HWPRF that hits in a cache at distance 3.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x841F`, **ISNP_HIT_N4_HWPRF, Snoop hit in cache at distance 4, hardware instruction prefetch**

The counter counts each snoop generated by an instruction hardware prefetch counted by ISNP_HIT_HWPRF that hits in a cache at distance 4.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x8420`, **DSNP_HIT_N1_PRF, Snoop hit in cache at distance 1, data prefetch**

The counter counts each snoop generated by the PE in response to a data prefetch instruction, or by a data hardware prefetch counted by DSNP_HIT_PRF that hits in a cache at distance 1.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x8421`**, DSNP_HIT_N2_PRF, Snoop hit in cache at distance 2, data prefetch**

The counter counts each snoop generated by the PE in response to a data prefetch instruction, or by a data hardware prefetch counted by DSNP_HIT_PRF that hits in a cache at distance 2.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x8422`**, DSNP_HIT_N3_PRF, Snoop hit in cache at distance 3, data prefetch**

The counter counts each snoop generated by the PE in response to a data prefetch instruction, or by a data hardware prefetch counted by DSNP_HIT_PRF that hits in a cache at distance 3.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x8423`**, DSNP_HIT_N4_PRF, Snoop hit in cache at distance 4, data prefetch**

The counter counts each snoop generated by the PE in response to a data prefetch instruction, or by a data hardware prefetch counted by DSNP_HIT_PRF that hits in a cache at distance 4.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x8424`**, ISNP_HIT_N1_PRF, Snoop hit in cache at distance 1, instruction prefetch**

The counter counts each snoop generated by the PE in response to an instruction prefetch instruction, or by an instruction hardware prefetch counted by ISNP_HIT_PRF that hits in a cache at distance 1.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x8425`**, ISNP_HIT_N2_PRF, Snoop hit in cache at distance 2, instruction prefetch**

The counter counts each snoop generated by the PE in response to an instruction prefetch instruction, or by an instruction hardware prefetch counted by ISNP_HIT_PRF that hits in a cache at distance 2.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x8426`**, ISNP_HIT_N3_PRF, Snoop hit in cache at distance 3, instruction prefetch**

The counter counts each snoop generated by the PE in response to an instruction prefetch instruction, or by an instruction hardware prefetch counted by ISNP_HIT_PRF that hits in a cache at distance 3.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x8427`**, ISNP_HIT_N4_PRF, Snoop hit in cache at distance 4, instruction prefetch**

The counter counts each snoop generated by the PE in response to an instruction prefetch instruction, or by an instruction hardware prefetch counted by ISNP_HIT_PRF that hits in a cache at distance 4.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x8428`**, ISNP_HIT_N1, Snoop hit in cache at distance 1, instruction access**

The counter counts each snoop counted by ISNP_HIT that hits in a cache at distance 1.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x8429`**, ISNP_HIT_N2, Snoop hit in cache at distance 2, instruction access**

The counter counts each snoop counted by ISNP_HIT that hits in a cache at distance 2.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x842A`**, ISNP_HIT_N3, Snoop hit in cache at distance 3, instruction access**

The counter counts each snoop counted by ISNP_HIT that hits in a cache at distance 3.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x842B`**, ISNP_HIT_N4, Snoop hit in cache at distance 4, instruction access**

The counter counts each snoop counted by ISNP_HIT that hits in a cache at distance 4.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x842C`, **DSNP_HIT_N1, Snoop hit in cache at distance 1, data access**

The counter counts each snoop counted by DSNP_HIT that hits in a cache at distance 1.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x842D`, **DSNP_HIT_N2, Snoop hit in cache at distance 2, data access**

The counter counts each snoop counted by DSNP_HIT that hits in a cache at distance 2.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x842E`, **DSNP_HIT_N3, Snoop hit in cache at distance 3, data access**

The counter counts each snoop counted by DSNP_HIT that hits in a cache at distance 3.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x842F`, **DSNP_HIT_N4, Snoop hit in cache at distance 4, data access**

The counter counts each snoop counted by DSNP_HIT that hits in a cache at distance 4.

The event is counted by the PE generating the snoop, not the PE being snooped.

The cache distance indicates the relative distance that the return data has traveled. The distance is a relative distance that is defined at the system level, allowing flexibility of use for different systems that may have differing levels of hierarchy.

The interpretation of distance is IMPLEMENTATION DEFINED, and should be the same for all PEs in the base system, and the same for both cache and memory distances.

`0x8431`, **ASE_FP_VREDUCE_SPEC, Floating-point operation speculatively executed, Advanced SIMD pairwise or reduction**

The counter counts each speculatively executed floating-point treewise reduction or pairwise operation counted by ASE_SVE_FP_VREDUCE_SPEC due to any of the following A64 instructions:

- Advanced SIMD: FADDP, FMAXNMP, FMAXNMV, FMAXP, FMAXV, FMINNMP, FMINNMV, FMINP, or FMINV.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Treewise reduction instructions might be executed and counted as multiple pairwise operations.

`0x8432`, **SVE_FP_PREDUCE_SPEC, Floating-point operation speculatively executed, SVE pairwise add step or pairwise reduce step**

The counter counts each speculatively executed floating-point pairwise add step or pairwise reduce step operation counted by SVE_FP_VREDUCE_SPEC due to any of the following instructions:

`0x8434`, **DSNP_HITM_N1_RD, Snoop hit in cache at distance 1, demand data read, modified**

The counter counts each snoop generated by the PE in response to a demand Memory-read operation counted by DSNP_HIT_N1_RD that accesses a previously modified cache location.

`0x8435`, **DSNP_HITM_N2_RD, Snoop hit in cache at distance 2, demand data read, modified**

The counter counts each snoop generated by the PE in response to a demand Memory-read operation counted by DSNP_HIT_N2_RD that accesses a previously modified cache location.

`0x8436`, **DSNP_HITM_N3_RD, Snoop hit in cache at distance 3, demand data read, modified**

The counter counts each snoop generated by the PE in response to a demand Memory-read operation counted by DSNP_HIT_N3_RD that accesses a previously modified cache location.

`0x8437`, **DSNP_HITM_N4_RD, Snoop hit in cache at distance 4, demand data read, modified**

The counter counts each snoop generated by the PE in response to a demand Memory-read operation counted by DSNP_HIT_N4_RD that accesses a previously modified cache location.

`0x8438`, **DSNP_HITM_N1_WR, Snoop hit in cache at distance 1, demand data write, modified**

The counter counts each snoop generated by the PE in response to a demand Memory-write operation counted by DSNP_HIT_N1_WR that accesses a previously modified cache location.

`0x8439`, **DSNP_HITM_N2_WR, Snoop hit in cache at distance 2, demand data write, modified**

The counter counts each snoop generated by the PE in response to a demand Memory-write operation counted by DSNP_HIT_N2_WR that accesses a previously modified cache location.

`0x843A`, **DSNP_HITM_N3_WR, Snoop hit in cache at distance 3, demand data write, modified**

The counter counts each snoop generated by the PE in response to a demand Memory-write operation counted by DSNP_HIT_N3_WR that accesses a previously modified cache location.

`0x843B`, **DSNP_HITM_N4_WR, Snoop hit in cache at distance 4, demand data write, modified**

The counter counts each snoop generated by the PE in response to a demand Memory-write operation counted by DSNP_HIT_N4_WR that accesses a previously modified cache location.

`0x843C`, **DSNP_HITM_N1_RW, Snoop hit in cache at distance 1, demand data access, modified**

The counter counts each snoop generated by the PE in response to a demand Memory-read operation or demand Memory-write operation counted by DSNP_HIT_N1_RW that accesses a previously modified cache location.

`0x843D`, **DSNP_HITM_N2_RW, Snoop hit in cache at distance 2, demand data access, modified**

The counter counts each snoop generated by the PE in response to a demand Memory-read operation or demand Memory-write operation counted by DSNP_HIT_N2_RW that accesses a previously modified cache location.

`0x843E`, **DSNP_HITM_N3_RW, Snoop hit in cache at distance 3, demand data access, modified**

The counter counts each snoop generated by the PE in response to a demand Memory-read operation or demand Memory-write operation counted by DSNP_HIT_N3_RW that accesses a previously modified cache location.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x843F`, **DSNP_HITM_N4_RW, Snoop hit in cache at distance 4, demand data access, modified**

The counter counts each snoop generated by the PE in response to a demand Memory-read operation or demand Memory-write operation counted by DSNP_HIT_N4_RW that accesses a previously modified cache location.

`0x8440`, **ASE_FP_SP_MIN_SPEC, Advanced SIMD data processing operation speculatively executed, smallest type is single-precision floating-point**

The counter counts each Advanced SIMD data-processing operation counted by ASE_SVE_FP_SP_MIN_SPEC due to an instruction where the smallest type was single-precision floating-point.

`0x8442`, **ASE_FP_HP_MIN_SPEC, Advanced SIMD data processing operation speculatively executed, smallest type is half-precision floating-point**

The counter counts each Advanced SIMD data-processing operation counted by ASE_SVE_FP_HP_MIN_SPEC due to an instruction where the smallest type was half-precision floating-point.

`0x8443`, **ASE_FP_BF16_MIN_SPEC, Advanced SIMD data processing operation speculatively executed, smallest type is BFloat16 floating-point**

The counter counts each Advanced SIMD data-processing operation counted by ASE_SVE_FP_BF16_MIN_SPEC due to an instruction where the smallest type was BFloat16 floating-point.

`0x8444`, **ASE_FP_FP8_MIN_SPEC, Advanced SIMD data processing operation speculatively executed, smallest type is 8-bit floating-point**

The counter counts each Advanced SIMD data-processing operation counted by ASE_SVE_FP_FP8_MIN_SPEC due to an instruction where the smallest type was 8-bit floating-point.

`0x8448`, **ASE_SVE_FP_SP_MIN_SPEC, Advanced SIMD data processing or SVE data processing operation speculatively executed, smallest type is single-precision floating-point**

The counter counts each Advanced SIMD or SVE data-processing operation counted by SE_FP_SP_MIN_SPEC due to an instruction where the smallest type was single-precision floating-point.

`0x844A`, **ASE_SVE_FP_HP_MIN_SPEC, Advanced SIMD data processing or SVE data processing operation speculatively executed, smallest type is half-precision floating-point**

The counter counts each Advanced SIMD or SVE data-processing operation counted by SE_FP_HP_MIN_SPEC due to an instruction where the smallest type was half-precision floating-point.

`0x844B`, **ASE_SVE_FP_BF16_MIN_SPEC, Advanced SIMD data processing or SVE data processing operation speculatively executed, smallest type is BFloat16 floating-point**

The counter counts each Advanced SIMD or SVE data-processing operation counted by SE_FP_BF16_MIN_SPEC due to an instruction where the smallest type was BFloat16 floating-point.

`0x844C`, **ASE_SVE_FP_FP8_MIN_SPEC, Advanced SIMD data processing or SVE data processing operation speculatively executed, smallest type is 8-bit floating-point**

The counter counts each Advanced SIMD or SVE data-processing operation counted by SE_FP_FP8_MIN_SPEC due to an instruction where the smallest type was 8-bit floating-point.

`0x8450`, **SE_FP_SP_MIN_SPEC, Advanced SIMD, SVE, or SME data processing operation speculatively executed, smallest type is single-precision floating-point**

The counter counts each Advanced SIMD, SVE, or SME data processing operation counted by FP_SP_MIN_SPEC due to an instruction where the smallest type was single-precision floating-point.

`0x8452`, **SE_FP_HP_MIN_SPEC, Advanced SIMD, SVE, or SME data processing operation speculatively executed, smallest type is half-precision floating-point**

The counter counts each Advanced SIMD, SVE, or SME data processing operation counted by FP_HP_MIN_SPEC due to an instruction where the smallest type was half-precision floating-point.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0x8453`, **SE_FP_BF16_MIN_SPEC, Advanced SIMD, SVE, or SME data processing operation speculatively executed, smallest type is BFloat16 floating-point**

The counter counts each Advanced SIMD, SVE, or SME data processing operation counted by FP_BF16_MIN_SPEC due to an instruction where the smallest type was BFloat16 floating-point.

`0x8454`, **SE_FP_FP8_MIN_SPEC, Advanced SIMD, SVE, or SME data processing operation speculatively executed, smallest type is 8-bit floating-point**

The counter counts each Advanced SIMD, SVE, or SME data processing operation counted by FP_FP8_MIN_SPEC due to an instruction where the smallest type was 8-bit floating-point.

`0x8458`, **SME_FP_SP_MIN_SPEC, SME data processing operation speculatively executed, smallest type is single-precision floating-point**

The counter counts each SME data-processing operation counted by SE_FP_SP_MIN_SPEC due to an instruction where the smallest type was single-precision floating-point.

`0x845A`, **SME_FP_HP_MIN_SPEC, SME data processing operation speculatively executed, smallest type is half-precision floating-point**

The counter counts each SME data-processing operation counted by SE_FP_HP_MIN_SPEC due to an instruction where the smallest type was half-precision floating-point.

`0x845B`, **SME_FP_BF16_MIN_SPEC, SME data processing operation speculatively executed, smallest type is BFloat16 floating-point**

The counter counts each SME data-processing operation counted by SE_FP_BF16_MIN_SPEC due to an instruction where the smallest type was BFloat16 floating-point.

`0x845C`, **SME_FP_FP8_MIN_SPEC, SME data processing operation speculatively executed, smallest type is 8-bit floating-point**

The counter counts each SME data-processing operation counted by SE_FP_FP8_MIN_SPEC due to an instruction where the smallest type was 8-bit floating-point.

`0x8460`, **SVE_FP_SP_MIN_SPEC, SVE data processing operation speculatively executed, smallest type is single-precision floating-point**

The counter counts each SVE data-processing operation counted by ASE_SVE_FP_SP_MIN_SPEC due to an instruction where the smallest type was single-precision floating-point.

`0x8462`, **SVE_FP_HP_MIN_SPEC, SVE data processing operation speculatively executed, smallest type is half-precision floating-point**

The counter counts each SVE data-processing operation counted by ASE_SVE_FP_HP_MIN_SPEC due to an instruction where the smallest type was half-precision floating-point.

`0x8463`, **SVE_FP_BF16_MIN_SPEC, SVE data processing operation speculatively executed, smallest type is BFloat16 floating-point**

The counter counts each SVE data-processing operation counted by ASE_SVE_FP_BF16_MIN_SPEC due to an instruction where the smallest type was BFloat16 floating-point.

`0x8464`, **SVE_FP_FP8_MIN_SPEC, SVE data processing operation speculatively executed, smallest type is 8-bit floating-point**

The counter counts each SVE data-processing operation counted by ASE_SVE_FP_FP8_MIN_SPEC due to an instruction where the smallest type was 8-bit floating-point.

`0x846B`, **FP_BF16_FIXED_OPS_SPEC, Non-scalable element arithmetic operations speculatively executed, largest type is BFloat16 floating-point**

The counter increments by v for each speculatively executed non-scalable element arithmetic operation, due to an instruction where the largest type was BFloat16 floating-point.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Where v is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

The counter does not count operations that are counted by FP_SCALE_OPS_SPEC.

`0x8470`, **FP_SP_MIN_SPEC, Floating-point operation speculatively executed, smallest type is single-precision floating-point**

The counter counts each floating-point operation counted by FP_SPEC due to an instruction where the smallest type was single-precision floating-point.

`0x8472`, **FP_HP_MIN_SPEC, Floating-point operation speculatively executed, smallest type is half-precision floating-point**

The counter counts each floating-point operation counted by FP_SPEC due to an instruction where the smallest type was half-precision floating-point.

`0x8473`, **FP_BF16_MIN_SPEC, Floating-point operation speculatively executed, smallest type is BFloat16 floating-point**

The counter counts each floating-point operation counted by FP_SPEC due to an instruction where the smallest type was BFloat16 floating-point.

`0x8474`, **FP_FP8_MIN_SPEC, Floating-point operation speculatively executed, smallest type is 8-bit floating-point**

The counter counts each floating-point operation counted by FP_SPEC due to an instruction where the smallest type was 8-bit floating-point.

`0x8480`, **FP_SP_FIXED_MIN_OPS_SPEC, Non-scalable element arithmetic operations speculatively executed, smallest type is single-precision floating-point**

The counter increments by v for each speculatively executed non-scalable element arithmetic operation, due to an instruction where the smallest type was single-precision floating-point.

Where v is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

The counter does not count operations that are counted by FP_SCALE_OPS_SPEC.

`0x8482`, **FP_HP_FIXED_MIN_OPS_SPEC, Non-scalable element arithmetic operations speculatively executed, smallest type is half-precision floating-point**

The counter increments by v for each speculatively executed non-scalable element arithmetic operation, due to an instruction where the smallest type was half-precision floating-point.

Where v is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

The counter does not count operations that are counted by FP_SCALE_OPS_SPEC.

`0x8483`, **FP_BF16_FIXED_MIN_OPS_SPEC, Non-scalable element arithmetic operations speculatively executed, smallest type is BFloat16 floating-point**

The counter increments by v for each speculatively executed non-scalable element arithmetic operation, due to an instruction where the smallest type was BFloat16 floating-point.

Where v is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

The counter does not count operations that are counted by FP_SCALE_OPS_SPEC.

`0x8484`, **FP_FP8_FIXED_MIN_OPS_SPEC, Non-scalable element arithmetic operations speculatively executed, smallest type is 8-bit floating-point**

The counter increments by v for each speculatively executed non-scalable element arithmetic operation, due to an instruction where the smallest type was 8-bit floating-point.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Where v is the number of arithmetic operations carried out by the operation or instruction which causes the counter to increment.

The counter does not count operations that are counted by FP_SCALE_OPS_SPEC.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D9.4 Cycle event counting

The CPU_CYCLES event and the cycle counter, PMCCNTR_EL0, Performance Monitors Cycle Count Register, count cycles. The duration of a cycle is subject to any changes in clock frequency, including clock stopping caused by the WFI and WFE instructions.

It is IMPLEMENTATION SPECIFIC whether CPU_CYCLES and PMCCNTR_EL0, Performance Monitors Cycle Count Register count when the PE is in WFI or WFE state, even if the clocks are not stopped.

In addition, events such as STALL, STALL_FRONTEND and STALL_BACKEND that are defined to only count cycles that are counted by the CPU_CYCLES event have the same limitation.

D9.4.1 Cycle event counting in multithreaded implementations

Multithreaded implementations can have various forms, some examples of these are:

- *Simultaneous Multithreading* (SMT), where every PE thread is active on every Processor cycle.
- *Fine-grained Multithreading* (FGMT), also known as a Barrel processor, where one PE thread is active on each Processor cycle, and this changes regularly.
- *Switch on Event Multithreading* (SoEMT), also known as *Coarse-grained Multithreading* (CGMT), where high latency events cause the processor to switch the active PE thread.

In the above examples, active means that the PE might execute the instructions. A PE can be active but not executing instructions when no instruction is available or because of limited execution resources.

It is IMPLEMENTATION SPECIFIC whether a thread is active when the thread is in WFE or WFI state. This applies for all forms of multithreaded implementation.

When the PMU implementation supports multithreading, the Effective value of PMEVTYPER<n>_EL0.MT bit is 0, and the thread is not in WFE or WFI state, the CPU_CYCLES event only counts Processor cycles when the thread is active. For the example multithreaded implementations, this means that, if the event counter is enabled, event counting is not prohibited, and the thread is not in WFE or WFI state:

- For an SMT implementation, the CPU_CYCLES event counts every Processor cycle.
- For a particular FGMT implementation, that alternates between two threads on each Processor cycle, the CPU_CYCLES event counts every other Processor cycle.
- For a particular SoEMT implementation, that is waiting for a long latency operation, the CPU_CYCLES event does not count Processor cycles, as the PE thread is not active.

It is IMPLEMENTATION SPECIFIC whether CPU_CYCLES event counts Processor cycles when the Effective value of PMEVTYPER<n>_EL0.MT bit is 0, the thread is active, and the thread is in WFE or WFI state.

If the Effective value of PMEVTYPER<n>_EL0.MT bit is 1 and at least one thread is not in WFE or WFI state, then the CPU_CYCLES event counts each Processor cycle, and can only count a maximum of one for each Processor cycle.

If the Effective value of PMEVTYPER<n>_EL0.MT bit is 1, then all of the following apply for events that count PE cycles when a condition applies:

- The event does not count Processor cycles when all threads are in WFE or WFI state.
- If the event counts PE cycles when a stall condition is true and a second condition is true, then the counter counts Processor cycles when the stall condition is true for all of these PEs, and the second condition is true for any of these PEs.
- If the event counts PE cycles when any other condition is true, then the counter counts Processor cycle when the condition is true for any of these PEs.
- Otherwise, the event counts by the sum of the count across all of these PEs.

For the stall events, the stall condition means the applicable condition described by the STALL, STALL_FRONTEND, or STALL_BACKEND event. The second condition is any condition in addition to this. For example, for the STALL_FRONTEND_L1I event, the stall condition is STALL_FRONTEND, and the second condition is when there is a demand instruction miss in the first level of instruction cache. For the STALL, STALL_FRONTEND, and STALL_BACKEND events themselves, the second condition is the null TRUE condition.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

The cycle counter, PMCCNTR_EL0, Performance Monitors Cycle Count Register, is not affected by whether the thread is active or inactive. When enabled, PMCCNTR_EL0, Performance Monitors Cycle Count Register counts every processor cycle.

See Multithreaded implementationsMDCR_EL3.MTPME for more information about when the Effective value of PMEVTYPER<n>_EL0.MT is 0.

D9.4.2 Time as measured by the Performance Monitors cycle counter

The Performance Monitors cycle counter, accessed through PMCCNTR_EL0 increments from the hardware processor clock, not PE clock cycles.

The relationship between the count recorded by the Performance Monitors cycle counter and the passage of real time is IMPLEMENTATION DEFINED.

See Controlling the PMU counters for information about when the cycle counter does not increment.

Note

- This means that, in an implementation where PEs are multithreaded, when enabled, the cycle counter continues to increment across all PEs, rather than only counting cycles for which the current PE is active.
- Although the architecture requires that direct reads of PMCCNTR_EL0 occur in program order, there is no requirement that the count increments between two such reads. Even when the counter is incrementing on every clock cycle, software might need check that the difference between two reads of the counter is nonzero.

  The architecture requires that an indirect write to the PMCCNTR_EL0 is observable to direct reads of the register in finite time. The counter increments from the hardware processor clock are indirect writes to these registers.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D9.5 Meaningful combinations of common events

The architecture highlights some meaningful combinations of common events. The following tables list the highlighted combinations:

- Table D9-4, REFILL events and associated access events.
- Table D9-5, Meaningful combinations of latency events.
- Table D9-6, Cache hit events and associated data source events.
- Table D9-7, Meaningful combinations of cache line state tracking events.
- Table D9-8, Meaningful combinations of TLB events.
- Table D9-9, Meaningful combinations of at-retirement events.
- Bytes loaded and stored events.

D9.5.1 Refill events and associated access events

I_{GSMLS} The following are meaningful combinations of refill events and associated access events:

Table D9-4 Meaningful combinations of refill events and associated access events

Numerator	Denominator	Ratio
0x0001 L1I_CACHE_REFILL	0x0014 L1I_CACHE	Attributable Level 1 instruction cache refill rate
0x0002 L1I_TLB_REFILL	0x0026 L1I_TLB	Attributable Level 1 instruction TLB refill rate
0x0003 L1D_CACHE_REFILL	0x0004 L1D_CACHE	Attributable Level 1 data or unified cache refill rate
0x0005 L1D_TLB_REFILL	0x0025 L1D_TLB	Attributable Level 1 data or unified TLB refill rate
0x0017 L2D_CACHE_REFILL	0x0016 L2D_CACHE	Attributable Level 2 data or unified cache refill rate
0x0028 L2I_CACHE_REFILL	0x0027 L2I_CACHE	Attributable Level 2 instruction cache refill rate
0x002A L3D_CACHE_REFILL	0x002B L3D_CACHE	Attributable Level 3 data or unified cache refill rate
0x002D L2D_TLB_REFILL	0x002F L2D_TLB	Attributable Level 2 data or unified TLB refill rate
0x002E L2I_TLB_REFILL	0x0030 L2I_TLB	Attributable Level 2 instruction TLB refill rate
0x0019 BUS_ACCESS	0x001D BUS_CYCLES	Attributable Bus accesses per cycle
0x0033 LL_CACHE_MISS	0x0032 LL_CACHE	Attributable Last Level data or unified cache refill rate
0x0034 DTLB_WALK	0x0025 L1D_TLB	Attributable data TLB miss rate
0x0035 ITLB_WALK	0x0026 L1I_TLB	Attributable instruction TLB miss rate
0x0037 LL_CACHE_MISS_RD	0x0036 LL_CACHE_RD	Attributable memory read operation miss rate
0x0038 REMOTE_ACCESS_RD	0x0031 REMOTE_ACCESS	Attributable read accesses to another socket in a multi-socket system
0x0042 L1D_CACHE_REFILL_RD	0x0040 L1D_CACHE_RD	Attributable Level 1 cache refill rate, read
0x0043 L1D_CACHE_REFILL_WR	0x0041 L1D_CACHE_WR	Attributable Level 1 cache refill rate, write
0x004C L1D_TLB_REFILL_RD	0x004E L1D_TLB_RD	Attributable Level 1 TLB refill rate, read
0x004D L1D_TLB_REFILL_WR	0x004F L1D_TLB_WR	Attributable Level 1 TLB refill rate, write
0x0052 L2D_CACHE_REFILL_RD	0x0050 L2D_CACHE_RD	Attributable Level 2 data cache refill rate, read
0x0053 L2D_CACHE_REFILL_WR	0x0051 L2D_CACHE_WR	Attributable Level 2 data cache refill rate, write
0x005C L2D_TLB_REFILL_RD	0x005E L2D_TLB_RD	Attributable Level 2 data TLB refill rate, read

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Numerator	Denominator	Ratio
0x005D L2D_TLB_REFILL_WR	0x005F L2D_TLB_WR	Attributable Level 2 data TLB refill rate, write
0x00A2 L3D_CACHE_REFILL_RD	0x00A0 L3D_CACHE_RD	Attributable Level 3 data cache refill rate, read
0x00A3 L3D_CACHE_REFILL_WR	0x00A1 L3D_CACHE_WR	Attributable Level 3 data cache refill rate, write

D9.5.2 Latency events

I_{ZHFDC} The following are meaningful combination of latency events:

Table D9-5 Meaningful combinations of latency events

Numerator	Denominator	Ratio
0x8120 INST_FETCH_PERCYC	0x8124 INST_FETCH	Mean duration of instruction fetch events in processor cycles
0x8121 MEM_ACCESS_RD_PERCYC	0x0066 MEM_ACCESS_RD	Mean duration of memory read access events in processor cycles
0x8125, BUS_REQ_RD_PERCYC	0x818D, BUS_REQ_RD	Bus read transaction average latency
0x8128 DTLB_WALK_PERCYC	0x0034 DTLB_WALK	Mean duration of data or unified TLB walk events in processor cycles
0x8129 ITLB_WALK_PERCYC	0x0035 ITLB_WALK	Mean duration of instruction TLB walk events in processor cycles
0x8320, L1D_CACHE_REFILL_PERCYC	0x0003, L1D_CACHE_REFILL	Level 1 data cache refill average duration

D9.5.3 Cache hit events and associated data source events

I_{HMZQY} The following are meaningful combination of cache hit events:

Table D9-6 Meaningful combinations of cache hit events and associated data source events

Numerator	Denominator	Ratio
0x820C, L1D_CACHE_HIT_PRFM [a]	0x8142, L1D_CACHE_PRFM [a]	Level 1 data or unified cache software prefetch hit ratio[a]
0x81CC, L1D_CACHE_HIT_RW [a]	0x82A0, MEM_ACCESS_RW	Level 1 data or unified cache demand cache hit rate[a]
0x81CF, LL_CACHE_HIT_RW	0x82A0, MEM_ACCESS_RW	Last level cache demand cache hit rate
0x8232, LOCAL_MEM_RW	0x82A0, MEM_ACCESS_RW	Local memory demand access rate
0x823B, REMOTE_MEM_RW	0x82A0, MEM_ACCESS_RW	Remote memory demand access rate
0x821D, L2D_CACHE_HITM_RW [a]	0x81CD, L2D_CACHE_HIT_RW [a]	Level 2 data or unified cache demand cache hit rate[a]

a. Similar ratios can be defined for other caches.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D9.5.4 Cache line state tracking events

I_{LGBBR} The following are meaningful combination of cache line stake tracking events:

Table D9-7 Meaningful combinations of cache line state tracking events

Ratio	Meaning
1 - (0x81EC, L1D_CACHE_HIT_RW_FHWPRF ÷ 0x8154, L1D_CACHE_HWPRF)[a]	Level 1 data or unified cache data fetched by hardware prefetcher not used[a]
1 - (0x81DC, L1D_CACHE_HIT_RW_FPRFM ÷ 0x8142, L1D_CACHE_PRFM)[a]	Level 1 data or unified cache data fetched by software prefetch not used[a]
0x825C, L1D_LFB_HIT_RW_FPRFM ÷ 0x8142, L1D_CACHE_PRFM [a]	Level 1 data or unified cache data fetched late by software prefetch[a]

a. Similar ratios can be defined for other caches.

D9.5.5 TLB events

I_{PHZQV} The following are meaningful combination of TLB events:

Table D9-8 Meaningful combinations of TLB events

Numerator	Denominator	Ratio
0x8188, DTLB_WALK_BLOCK [a]	0x0034, DTLB_WALK [a]	Data TLB block translation table walk fraction[a]
0x818A, DTLB_WALK_PAGE [a]	0x0034, DTLB_WALK [a]	Data TLB page translation table walk fraction[a]

a. Similar ratios can be defined for instruction TLB accesses.

D9.5.6 At-retirement events

I_{CYQZJ} The following are meaningful combinations of at-retirement events:

Table D9-9 Meaningful combinations of at-retirement events

Numerator	Denominator	Ratio
0x82B0, L1I_CACHE_HIT_RETIRED	0x0008, INST_RETIRED	Architecturally executed instruction hit Level 1 instruction cache
0x82E0, L2I_CACHE_MISS_RETIRED	0x0008, INST_RETIRED	Architecturally executed instruction missed Level 2 instruction cache
0x82EC, L1D_CACHE_HIT_LDST_RETIRED	0x0008, INST_RETIRED	Architecturally executed load or store instruction hit in Level 1 data cache
0x82C9, L2D_CACHE_HITM_ST_RETIRED	0x0008, INST_RETIRED	Architecturally executed store instruction hit modified data in Level 2 data cache
0x82B7, LL_CACHE_MISS_LD_RETIRED	0x0008, INST_RETIRED	Architecturally executed load instruction missed in Last level cache

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Numerator	Denominator	Ratio
0x82F0, ITLB_HIT_RETIRED	0x0008, INST_RETIRED	Architecturally executed instruction which hit in the ITLB
0x82F7, DTLB_WALK_LDST_RETIRED	0x82AA, LDST_ANY_RETIRED	Architecturally executed LDST instructions DTLB miss fraction
0x82AF, CRYPTO_RETIRED	0x0008, INST_RETIRED	Architecturally executed cryptographic data processing fraction

D9.5.7 Speculatively executed operation events

I_{RDHPB} The number of speculatively executed operations performed on individual scalar values can be determined from a pair of event counters. Combined multiply-add and multiply-subtract instructions are counted as two operations per element.

Table D9-10 Meaningful combinations of speculatively executed operation events

Ratio	Meaning
0x80C0, FP_SCALE_OPS_SPEC × VL ÷ 128 + 0x80C1, FP_FIXED_OPS_SPEC	Total number of individual floating-point operations (any precision) performed
0x80C2, FP_HP_SCALE_OPS_SPEC × VL ÷ 128 + 0x80C3, FP_HP_FIXED_OPS_SPEC	Total number of individual half-precision floating-point operations performed
0x80C4, FP_SP_SCALE_OPS_SPEC × VL ÷ 128 + 0x80C5, FP_SP_FIXED_OPS_SPEC	Total number of individual single-precision floating-point operations performed
0x80C6, FP_DP_SCALE_OPS_SPEC × VL ÷ 128 + 0x80C7, FP_DP_FIXED_OPS_SPEC	Total number of individual double-precision floating-point operations performed
0x80C8, INT_SCALE_OPS_SPEC × VL ÷ 128 + 0x80C9 INT_FIXED_OPS_SPEC	Total number of individual integer operations performed
0x80CA, LDST_SCALE_OPS_SPEC × VL ÷ 128 + 0x80CB, LDST_FIXED_OPS_SPEC	Total number of individual load/store accesses performed
0x80CC, LD_SCALE_OPS_SPEC × VL ÷ 128 + 0x80CD, LD_FIXED_OPS_SPEC	Total number of individual load accesses performed
0x80CE, ST_SCALE_OPS_SPEC × VL ÷ 128 + 0x80CF, ST_FIXED_OPS_SPEC	Total number of individual store accesses performed

D9.5.8 Bytes loaded and stored events

I_{NYJRH} The number of bytes speculatively loaded from memory or stored to memory can be determined from a pair of event counters.

Table D9-11 Meaningful combinations of speculative load or store events

Ratio	Meaning
0x80DA, LDST_SCALE_BYTES_SPEC × VL ÷ 128 + 0x80DB, LDST_FIXED_BYTES_SPEC	Total number of bytes loaded from memory and stored to memory
0x80DC, LD_SCALE_BYTES_SPEC × VL ÷ 128 + 0x80DD, LD_FIXED_BYTES_SPEC	Total number of bytes loaded from memory

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Ratio	Meaning
0x80DE, ST_SCALE_BYTES_SPEC × VL ÷ 128 + 0x80DF, ST_FIXED_BYTES_SPEC	Total number of bytes stored to memory

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D9.6 Required events

When at least one event counter is implemented, the following common events are implemented:

- 0x0000, SW_INCR, Instruction architecturally executed, Condition code check pass, software increment.
- 0x0003, L1D_CACHE_REFILL, Level 1 data cache refill.

  Note

  Event 0x0003 is only required if the implementation includes a Level 1 data or unified cache.

- 0x0004, L1D_CACHE, Level 1 data cache access.

  Note

  Event 0x0004 is only required if the implementation includes a Level 1 data or unified cache.

- 0x0010, BR_MIS_PRED, Mispredicted or not predicted branch Speculatively executed.

  Note

  Event 0x0010 is only required if the implementation includes program-flow prediction. However, Arm strongly recommends that the event is implemented as described in Common microarchitectural events.

- 0x0011, CPU_CYCLES, Cycle.
- 0x0012, BR_PRED, Predictable branch Speculatively executed.

  Note

  Event 0x0012 is only required if the implementation includes program-flow prediction. However, Arm recommends that the event is implemented as described in Common microarchitectural events.

- When FEAT_PMUv3_ICNTR is implemented:
  - 0x0008, INST_RETIRED, Instruction architecturally executed.
- When FEAT_PMUv3_ICNTR is not implemented, at least one of:
  - 0x0008, INST_RETIRED, Instruction architecturally executed.
  - 0x001B, INST_SPEC, Operation Speculatively executed.

  Note

  Arm strongly recommends that event 0x0008 is implemented.

- When FEAT_PMUv3p1 is implemented:
  - 0x0023, STALL_FRONTEND, No operation issued due to the frontend.
  - 0x0024, STALL_BACKEND, No operation issued due to the backend.

- When FEAT_PMUv3p4 is implemented:
  - 0x003C, STALL, No operation sent for execution.
  - 0x0039, L1D_CACHE_LMISS_RD, Level 1 data cache long-latency read miss.
  - 0x4006, L1I_CACHE_LMISS, Level 1 instruction cache long-latency miss.
  - 0x0040, L1D_CACHE_RD, Level 1 data cache read.

When any of the following common events are implemented, all three of them are implemented:

- 0x003D, STALL_SLOT_BACKEND, No operation sent for execution on a Slot due to the backend.
- 0x003E, STALL_SLOT_FRONTEND, No operation sent for execution on a Slot due to the frontend.
- 0x003F, STALL_SLOT, No operation sent for execution on a Slot.

Arm strongly recommends that the following events are implemented:

- 0x0021, BR_RETIRED.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- `0x0022`, BR_MIS_PRED_RETIRED.
- `0x003A`, OP_RETIRED.
- `0x003B`, OP_SPEC.
- `0x003D`, STALL_SLOT_BACKEND.
- `0x003E`, STALL_SLOT_FRONTEND.
- `0x003F`, STALL_SLOT.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D9.7 Synchronous events

Only events that count instructions at retirement can be synchronous events, but not all such events are required to be synchronous events.

An at-retirement event is an event generated by an architecturally-executed instruction that does not generate an exception.

Arm expects the set of synchronous events to include, but not be limited to:

- Existing events that count instructions at retirement:
  - INST_RETIRED.
  - Events that count instructions based on a taxonomy of instruction types.
  - Other existing branch events that count branches at retirement based on their behavior during execution.
- Events that count retired instructions that missed or hit in the instruction TLB or cache.
- Events that count retired load instructions that missed or hit in the data TLB or cache.

Events that are not attributable to an instruction, or are attributable to an instruction that generates an exception, such as EXC_TAKEN, are never synchronous events.

Note

Software requires *a priori* knowledge of which events are synchronous events. There is no hardware identification mechanism. A systematic software ID mechanism, such as a table in firmware, should be implemented.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D9.8 IMPLEMENTATION DEFINED event numbers

Arm recommends that implementers establish a standardized numbering scheme for their IMPLEMENTATION DEFINED events, with common definitions, and common event numbers, applied to all of their implementations. In general, the recommended approach is for standardization across implementations with common features. However, Arm recognizes that attempting to standardize the encoding of microarchitectural features across too wide a range of implementations is not productive.

The Arm architecture guarantees not to define any event prefixed with *IMP_* as part of the standard Arm architecture.

Arm strongly recommends that at least the following classes of event are identified in the IMPLEMENTATION DEFINED events:

- Separating each of the STALL_FRONTEND and STALL_SLOT_FRONTEND events to count holes in instruction availability.
- Separating each of the STALL_BACKEND and STALL_SLOT_BACKEND events, to count, for example, cumulative duration of stalls, unavailability of execution resources, or missed superscalar issue opportunities.
- Miss rates for additional levels of caches and TLBs.
- Any external events passed to the PE through an IMPLEMENTATION DEFINED mechanism.
- Cumulative duration of a PSTATE.{A, I, F} interrupt mask set to 1.
- Cumulative occupancy for resource queues, such as data access queues, and entry/exit counts, so that average latencies can be determined, separating out counts for key resources that might exist. An implementation might also provide registers in the IMPLEMENTATION DEFINED space to further extend such counts, for example by specifying a minimum latency for an event to be counted.
- Any other microarchitectural features that the implementer considers are valuable to count.

The range of possible IMPLEMENTATION DEFINED event numbers is described in The PMU event number space and common events.

It is not required that implementations document the behavior of all IMPLEMENTATION DEFINED PMU events, including which PMU event numbers are used for these IMPLEMENTATION DEFINED events. However, Arm strongly recommends that implementations do include and document IMPLEMENTATION DEFINED PMU events. In particular events related to microarchitectural characteristics of the implementation not covered by Common events.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter D10
RAS PE Architecture

This chapter describes the RAS Extension PE Architecture. It contains the following sections:

- About the RAS Extension.
- PE error handling.
- Generating error exceptions.
- Taking error exceptions.
- Error synchronization event.
- Virtual SError exceptions and delegated SError exceptions.
- Error records in the PE.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D10.1 About the RAS Extension

I_{LBKPL} The RAS Extension extends the exception syndrome registers to include fields that allow the PE to report a PE error state when an Error exception is taken.

I_{THGHB} The RAS Extension defines System registers that are specific to RAS, including to access optional Error records defined by the RAS System Architecture. The format of the error record registers is defined in Error record System register view.

R_{YWXWL} The RAS Extension does not prescribe the level of reliability, availability, and serviceability in the PE. The RAS features that the system and PE include, for example to detect, correct, contain, or defer errors, are IMPLEMENTATION DEFINED. The RAS Extension defines a framework for building RAS features in a PE.

Example D10-1 Common error handling for RAS features

For a system with a requirement for high reliability and error containment:

- When errors are detected, they are recorded in error records. This includes both correctable and uncorrectable errors. Error correction and detection logic exists on all significant memory structures, and might include other non-memory structures.
- Most uncorrectable errors generate poison values to defer the error.
- On consuming poison from the memory system, a load instruction that would corrupt PE state generates a synchronous External Abort exception, regardless of the memory type.
- On taking error exceptions, the PE error state is Recoverable state (UER).

I_{QGLBZ} See also *Arm® Reliability Availability and Serviceability (RAS) System Architecture, for A-profile architecture* (ARM IHI 0100) for more information about the framework for building RAS features in a system.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D10.2 PE error handling

D10.2.1 PE error detection

I_{KRYQW} When a PE accesses memory or other state, an error might be detected in that memory or state, and corrected, deferred, or signaled to the PE as a Detected error with an in-band error response.
Note:
- An error that is deferred might be signaled to the PE with an in-band error response.
- An error might also be signaled to a PE by means other than an in-band error response.

I_{SLYLK} The response from memory or other state is defined by Detecting and consuming errors. For more information, see *Arm® Reliability, Availability, and Serviceability (RAS) System Architecture, for A-profile architecture* (ARM IHI 0100).

I_{DWWQJ} When an error is detected by a component on a read or a cache maintenance operation from the PE:
- If the error can be corrected, it is corrected and corrected data is returned.
- If the error cannot be corrected and can be deferred, it is deferred. For example, on a load by poisoning the PE state, if this is supported by the PE implementation.
- If the error cannot be corrected and if implemented and enabled at the component, the Detected error is signaled to the PE as an in-band error response.

I_{BKVQP} When an error is detected by a component consuming a write from the PE:
- If the error can be corrected, it is corrected.
- If the error cannot be corrected and can be deferred, it is deferred to the consumer. For example, by poisoning the location being written.
- If the error cannot be corrected and if implemented and enabled at the component, the Detected error is signaled to the PE as an in-band error response.

I_{PDDNB} The component might record the Detected error and generate a Fault handling interrupt and/or Error recovery interrupt.

I_{VRYFF} If the component implements the RAS System Architecture, its behavior is defined by the RAS System Architecture, and depends on the nature of the error and IMPLEMENTATION DEFINED properties of the component. In each of these cases, the component might be a part of the processor, such as a cache, or might be outside of the processor.
The component might also report the error to a RAS System Architecture node, which records the error and might generate one or more of a Fault handling interrupt, Error recovery interrupt, or Critical Error interrupt depending on the features and configuration of the node.
See also Other errors.

I_{FLXTY} An in-band error response is sometimes referred to as an *External abort*. To avoid confusion with the External abort *exception*, the RAS sections in this Manual use in-band error response to describe the response to the PE for a memory access.
For more information, see *Arm® Reliability, Availability, and Serviceability (RAS) System Architecture, for A-profile architecture* (ARM IHI 0100).

R_{WLTPV} The size of the protection granule for any implemented error detection mechanism in memory is IMPLEMENTATION DEFINED and might vary from moment to moment and instruction to instruction.
For example, the size of the protection granule might depend on one or more of the following:
- The physical address of the Location being accessed.
- Whether a copy of the Location held in a cache is being accessed, and the cache it is held in.
- The type of access being made.
- If FEAT_SME is implemented, whether the PE is in Streaming SVE mode or is not in Streaming SVE mode.

I_{JJJGW} A system might implement multiple error detection mechanisms with differing protection granule sizes.

I_{YBRSM} For a Deferred error, Clearing Deferred errors from poisoned Locations describes mechanisms to remove poison and clear the error. The architecture also permits IMPLEMENTATION DEFINED mechanisms for clearing an error or poison from memory.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D10.2.2 PE error propagation

I_{RPLQT} The program-visible architectural state of the PE, referred to as the PE state, includes:

- General-purpose and SIMD&FP registers.
- System registers.
- Special-purpose registers.
- PSTATE.

R_{XMBNW} An error is consumed by the PE by any of the following:

- An instruction commits the corruption into the PE state.
- The error is on an instruction fetch and the corrupt instruction is committed for execution.
- The error is on a translation table walk for a committed load, store, or instruction fetch.

I_{HVFKW} For a PE, Error propagation applies to the propagation of Detected error by the PE between the PE state, and any other PE state or memory.

Note: *Memory* includes structures that cache the contents of memory, such as an instruction cache, data cache, or TLB.

R_{VGXBJ} An error is propagated by the PE by one or more of the following occurring that would not have been permitted to occur had the fault not been activated:

- Consumption of the corrupt value by any instruction, propagating the error to the target(s) of the instruction. This includes:
  - A store of a corrupt value.
  - A write of a corrupt value to a System register, Special-purpose register, or PSTATE. Infecting a System register state might mean that the PE generates transactions that would not otherwise be permitted.
- Any operation occurring that should not have occurred, including:
  - A load, translation table walk, or instruction fetch that would not have been permitted, including those from hardware speculation or prefetching.
  - A store to an incorrect address, or a store that would not have been made or not permitted.
  - A direct or indirect write to a Special-purpose or System register that would not have been made or not permitted.
  - Assertion of any signal, such as an interrupt, that would not have been asserted.
- Any operation not occurring that should have occurred.
- Causing the PE to take an imprecise exception, other than an Error exception in response to the error itself. For more information, see Definition of a precise exception and imprecise exception.
- The PE discarding data that it holds in a modified state.
- Any other loss of required uniprocessor semantics, ordering, or coherency.

I_{RMZNK} In R_{VGXBJ}, *not have been permitted to occur* means that the observable behavior of the PE is a deviation from the correct service of the PE, as defined by the architecture. Deviations from the normal behavior of the PE implementation that would otherwise be permitted by the architecture are not deviations from correct service.

Example D10-2

A PE takes an Error exception asynchronously as follows, in program order:

1. A load returns a corrupt value from a first location to a general-purpose register.
2. The PE suppresses a store of the register to a second memory location. In particular, the location is not updated and so retains its previous value.
3. The Error exception is taken.
4. At the point when the Error exception is taken, the ordering constraints imposed by the architecture have not been violated, in particular those relating to the observability of the store at step 2.

Although the error has not been propagated, the PE state is not consistent with the PE having executed all of the instructions up to the point when the Error exception is taken, and so it would be unlikely that software would be able to recover execution.

R_{NQDWB} An error propagated by the PE is silently propagated by the PE only if all of the following are true:

1. The propagation is not part of the required operation of the PE in taking an Error exception generated by the error.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2. The propagation is not part of the required operation of the PE executing an ESB instruction that synchronizes the error.
3. The error is not signaled to the consumer as a Detected error or Deferred error.
4. Any of the following are true:
   - The corrupt value is held in other than the general-purpose and SIMD&FP registers.
   - The error is propagated by an instruction in program order before either taking an Error exception generated by the error or executing an ESB instruction that synchronizes the error, and is propagated to outside of the general-purpose and SIMD&FP registers.
   - The error is propagated other than by an instruction that consumes the corrupt value as an input operand but otherwise behaves correctly.

I_{BRBFF} In R_{NQDWB}, item 4 means that after taking the Error exception generated by the error, or an ESB, propagating an error by, for example, storing the corrupt value to memory, is not considered as silent propagation of the error by the PE.

Example D10-3

A PE takes an Error exception in response to a load that returns a corrupt value to a general-purpose register. The error is not silently propagated to outside of the general-purpose registers before the Error exception is taken. Neither of the following are considered silent propagation of the error by the PE:

- Taking the Error exception causes the ESR_ELx, ELR_ELx, and SPSR_ELx registers to be updated. This is part of the required operation of the PE.
- After taking the Error exception, software stores the contents of the general-purpose register to memory, and this is not signaled to memory as a Deferred error. This happens in program order after the exception is taken.

The error is not silently propagated by the PE.

Example D10-4

Further to Example D10-2, if either of the following example additional operations occur between 2 and 3, then the PE has silently propagated the error:

- A second store to a third location is performed by the PE, and the architecture requires that the first store is ordered-before the second store. For example, the second store is a store-release operation. In this case, the PE violates the external ordering constraints for the two stores, and the error is silently propagated to any observer of the second store.
- A second load from the second location returns the previous value, and a second store writes that value to a third location. In this case, the PE violates the requirements of the Coherence-before relation because the second load cannot be Coherence-before the first store. The error silently propagates to any observer of the second store.

If instead of the PE suppressing the store at 2, the PE poisons the second memory location, and in the second example propagates that poison through the second load and second store, then the error is not silently propagated.

R_{DTRFQ} The features that a PE includes to prevent silent propagation of an error are IMPLEMENTATION DEFINED.

Example D10-5

An implementation ensures that a corrupt value in a general-purpose or SIMD&FP register is not silently propagated, by signalling a Deferred error on any write of data to any memory location, such that the memory location is poisoned.

D10.2.3 Clearing Deferred errors from poisoned Locations

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D10.2.3.1 Poison-atomicity of Memory Write Effects

D_{YQQLJ} A Memory Write Effect is *poison-atomic* if the Memory Write Effect removes a Deferred error (poison) at all Locations written to by the Memory Write Effect.

R_{XVFDG} For each store instruction with Explicit Memory Write Effects, it is IMPLEMENTATION DEFINED whether the Memory Write Effects are poison-atomic for a region of physical memory. This includes the `DC ZVA` and/or `DC GZVA` instructions.

R_{QSBKL} If a Location in a region of physical memory supports the poison-atomic property for a store instruction, then all Locations in the same naturally-aligned region support this for the same store instruction, where the size of the region is the largest translation granule that the PE supports.

I_{CSJLP} Support for the poison-atomic property for a store instruction is a property of both the PE as well as the system. That is, if the system includes a PE that supports the poison-atomic property for the instruction, then both the PE and the system support this instruction such that it is guaranteed to be poison-atomic.

For example, the system should not split cache line writes of data to be smaller than the poison granule of the downstream component in order to preserve poison-atomicity of a `DC {G}ZVA` instruction.

I_{ZRSSM} There is no architected means to discover the poison-atomic property for `DC ZVA` or `DC GZVA` for any region of physical memory.

I_{NXNTY} When a Memory Write Effect to a Location is not poison-atomic, the Memory Write Effect propagates a Deferred error previously at the Location to the new value being written.

Example D10-6

A region of physical memory uses Error Correction Codes (ECC) to provide error detection and correction. The protection granule for this region of memory is the size of a range covered by a single ECC. The ECC is also used to encode a poison value. This means the poison granule is the same size as the protection granule, for this region of physical memory.

A write that is smaller than the protection granule to this region of physical memory must read the protection granule containing the Location being written in order to compute a new ECC for the protection granule. If the protection granule contains poison, then a poisoned value is written back, meaning the write is not poison-atomic.

If a write overwrites the entire protection granule, then the system can compute a new ECC for the protection granule without reading the previous contents, meaning the write can be poison-atomic.

I_{GLFCV} A system might require software to stop using the protection granule until the Deferred error can be removed.

For example, by not using the physical page containing the granule until the system is reset. The architecture does not set any limit on the size of a protection granule, and it might be larger than a translation granule.

I_{HPJFJ} The size of the poison granule is IMPLEMENTATION DEFINED and might vary from moment to moment, instruction to instruction, and by Location. The size of the poison granule might be the same as the size of the protection granule, but this is not guaranteed. See also R_{WLTPV}.

I_{CBCMK} The regions of physical memory that support the poison-atomic property are IMPLEMENTATION DEFINED. Systems might have multiple physical memory types, including multiple physical memory regions used for bulk memory.

D10.2.3.2 Removing poison

I_{VZRKL} For guidance on mechanisms that software can use to remove poison from a Location, see Use of the RAS Extension.

I_{PSLGW} In order to test that poison has been removed from a Location, after the Location has been overwritten by a known-good value, software must be able to read back the Location from physical memory to check that the poison has been removed and the error is not persistent.

To do this, software must ensure that the Location has been cleaned and invalidated from all caches, so that the read returns a value from the Point of Physical Storage for the Location, and not any intermediate cache.

The Point of Coherency might be before a transparent unified cache, meaning that it might be before the Point of Physical Storage for a Location.

For more information on the Point of Physical Storage, see A64 Cache maintenance instructions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D10.2.4 Other errors

I_{KRQMR} The RAS Extension deals mostly with errors detected by components outside of the PE, such as memory, and consumed by the PE.

Other errors might be detected from within the processor that implements the PE. If the error is not an error in the PE state then the error might be treated as an error detected by another component.

In the following examples, the *component* reports these errors to a RAS System Architecture node that implements Error records and records the errors, and might generate one or more of a Fault handling interrupt, Error recovery interrupt, or Critical Error interrupt depending on the features and configuration of the node.

Example D10-7

A processor cache can be treated as a component outside the PE.
The cache detects an error in the cache state that cannot be corrected:

- If the error is detected in dirty cache data being evicted from the cache when the PE makes an access, then the error might be deferred by the cache writing poison in the evicted cache data.
- If the PE is performing a partial write that does not completely overwrite the protection granule, then the error might be deferred by the cache writing poisoned to the cache location, and/or evicting the cache line with poison. Deferring the error means the error is not consumed by the PE.

Otherwise, the cache component generates the in-band error response to the PE.

Example D10-8

A processor memory interface can be treated as a component outside the PE.
A processor detects a corrupt or poisoned value being returned on the interface that is not being signaled as an in-band error response and cannot be corrected or deferred. For example, in response to a non-cacheable read or a cache refill.
The memory interface component generates the in-band error response to the PE.

I_{VNTWD} An implementation might include error detection logic within the PE state itself. When the PE detects an error in the PE state, the instruction that uses that state consumes the error, and the PE generates an IMPLEMENTATION DEFINED Error exception, taken as an SError exception. See R_{FNVVJ}.

In this case, the processor that implements the PE includes a RAS System Architecture node that implements Error records that record these errors.

I_{JRQDM} An implementation might support poisoning within the PE state. When the PE consumes a Deferred error, for example a poisoned value, from memory into the PE state, the PE state becomes poisoned. Subsequent operations that read the poisoned value can continue to defer the error by poisoning the result of the operation.

However, if the PE attempts to execute an operation that reads the poisoned value and cannot defer the error further, the PE generates an IMPLEMENTATION DEFINED Error exception, taken as an SError exception. See R_{FNVVJ}.

In this case, the processor that implements the PE includes a RAS System Architecture node that implements Error records that record these errors.

I_{JHQVK} Components outside of the PE might detect errors that are not consumed by the PE. These components might report such errors to a PE using Error recovery interrupts.

I_{MJQQZ} Errors from software faults are outside the scope of the RAS Extension. For more information, see *Arm® Reliability, Availability, and Serviceability (RAS) System Architecture, for A-profile architecture* (ARM IHI 0100).

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D10.3 Generating error exceptions

R_{VJRKF} An Error exception is generated when a detected error is signaled to the PE as an in-band error response to an architecturally-executed memory access or cache maintenance operation. This includes any explicit data access, instruction fetch, translation table walk, or hardware update to the translation tables made by an architecturally-executed instruction.

I_{PJHZS} An Error exception is taken as an asynchronous SError exception, a synchronous External Data Abort exception, or a synchronous External Instruction Abort exception.

R_{MBNBH} It is IMPLEMENTATION DEFINED whether an Error exception can be generated for an error that is consumed by hardware speculation or prefetching by a PE, but that is not committed to the architecturally visible state of the PE.

R_{SHKJB} It is IMPLEMENTATION DEFINED whether an Error exception can be generated for a detected error that is deferred.

R_{GVWJD} It is IMPLEMENTATION DEFINED whether an Error exception can be generated for a detected error that is corrected.

R_{FNVVJ} An Error exception can also be generated for IMPLEMENTATION DEFINED causes. An Error exception generated for an IMPLEMENTATION DEFINED cause is taken as an SError exception.

Example D10-9

An error is detected and neither corrected nor deferred to the PE, and signaled to the PE by means other than an in-band error response, such as a wired SError exception pin. Asserting the SError exception pin causes the PE to generate an SError exception.

Example D10-10

An error is detected by the PE in the PE state, or in the result of a calculation performed by the PE. The detected error generates an SError exception.

I_{ZXHDP} See also:

- Exceptions.
- Taking error exceptions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D10.4 Taking error exceptions

R_{PKGLV} For an error signaled on a non-speculative instruction fetch or a translation table walk or hardware update of translation tables, it is IMPLEMENTATION DEFINED whether an Error exception on these accesses is taken as a synchronous External Abort exception or as an asynchronous SError exception.

R_{VLFTP} Unless otherwise stated, it is IMPLEMENTATION DEFINED whether an Error exception is handled precisely and synchronously, or handled asynchronously for each non-speculative:

- Explicit memory access made by an instruction.
- Cache maintenance operation.
- Translation table walk or hardware update of translation tables.

D_{SRYHT} In this section, *Handled precisely and synchronously* means the error generates a precise synchronous Data Abort exception.

D_{THFRY} In this section, *Handled asynchronously* means the error generates an asynchronous SError exception.

R_{CLFJM} All Error exceptions generated by speculative memory accesses are taken as an asynchronous SError exception.

R_{WFNJG} When an Error exception is taken as an asynchronous SError exception, the exception is taken in finite time.

R_{BCXKN} When any of the following exceptions are taken, the PE records the PE error state in the exception syndrome register:

- A synchronous External abort.
- An SError exception.

I_{GGCQQ} The exception type and target Execution state determine the set of PE error state values the PE can record.

I_{WSYXB} The recorded PE error state informs software whether software can recover execution and, if so, whether any action by the recovery software to locate and repair the error is necessary first.

I_{TRWXF} Other than as described by I_{WBVYC}, the PE error state recorded in the exception syndrome register describes the recovery of the PE state only. For example, the PE state might be recoverable when the state of system is such that system-level recovery is not possible. See also I_{ZQRGL}.

I_{NFDSM} Software is only able to successfully recover execution and make progress from a restart address for the exception by executing an Exception Return instruction to branch to the instruction at this restart address if all of the following are true:

- The error has not been silently propagated by the PE.
- At the point when the Exception Return instruction is executed, the PE state and memory system state are consistent with the PE having executed all of the instructions up to but not including the instruction at the restart address, and none afterwards. That is, at least one of the following restart conditions is true:
  - The error has been not architecturally consumed by the PE and infected the PE state.
  - Executing the instruction at the restart address will not consume the error and will correct any corrupt state by overwriting it with the correct value or values.

R_{DCKHJ} On taking an Error exception, the PE determines that software is able to recover execution at the point where the exception is taken, with no additional action from software, if and only if all of the following are true:

- The error has not been silently propagated by the PE.
- The restart conditions are met because all of the following are true:
  - Either the error does not remain latent or executing the instruction at the restart address will not consume the error and will correct any corrupt PE state.
  - The restart address is the preferred return address for the exception.
- The PE has not elected to determine that software is not able to recover execution, and has not elected to determine that software is able to recover execution if software takes action to locate and repair the error.

R_{JBHWY} On taking an Error exception, the PE determines that software is able to recover execution if software takes action to locate and repair the error, to get the PE state and memory system state into this consistent state before attempting recovery, if and only if all the following are true:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The error has not been silently propagated by the PE.
- The restart conditions can be met because the restart address is the preferred return address for the exception and at least one of the following is true:
  - The error remains latent and executing the instruction at the restart address will access the corrupt state. If the error is removed then executing the instruction at the restart address will correct any corrupt PE state and/or corrupt memory state. For example, the instruction at the restart address is a load that will consume the error and corrupts PE state.
  - The error does not remain latent and the PE has elected to determine that software is able to recover execution if software takes action to locate and repair the error.
  - Executing the instruction at the restart address will not consume the error and the PE has elected to determine that software is able to recover execution if software takes action to locate and repair the error.
- The PE has not elected to determine that software is not able to recover execution.

R_{GJQWN} On taking an Error exception, the PE determines that software is not able to recover execution if and only if one or more of the following are true:

- The error has been silently propagated by the PE.
- The restart conditions cannot be met even if software takes action to locate and repair the error. This is because at least one of the following is true:
  - The error remains latent and executing the instruction at the restart address will consume the error and corrupt PE state. Either the error cannot be removed or executing the instruction at the restart address will not correct any corrupt PE state.
  - The restart address is not the preferred return address for the exception.
- The PE has elected to determine that software is not able to recover execution.

I_{XMCCR} That the PE determines that software is able to recover execution if software takes action to locate and repair the error does not mean that software can locate and repair the error. For example, the error in memory might be one which cannot be located or cannot be repaired. See I_{ZQRGL}.
Error recovery software might instead make the PE state and memory system state consistent with an *alternative execution* of the program.

Example D10-11

An Error exception is generated by a load from a location in a clean page of memory that is infected by an error. Software might be able to repair the error by:

- Reloading the page from a backing store. This makes the memory system state consistent with the uncorrupted view. Executing the instruction at the restart address will load the uncorrupted value into the PE state.
- Invalidating the clean page and marking it page as inaccessible. Executing the instruction at the restart address will result in a Translation fault being generated when the program tries to access the page. The target of the load will contain an UNKNOWN value, which is permitted by the architecture. The MMU fault handler can then reload the page from the backing store, as it would for a page that has not been previously accessed or has been paged out.

Either approach might result in the virtual address to physical address mapping for the page being changed by software, meaning the memory system state is not consistent with the previously executed instructions. However, the memory system state is consistent with a valid alternative view of the execution of the program that allows software to recover execution.
This recovery is only possible if the error can be isolated to a location.

I_{RHPPV} A PE might include additional IMPLEMENTATION DEFINED mechanisms to aid software locate and repair the error.
If software has to use IMPLEMENTATION DEFINED mechanisms to locate and repair the error, then the PE reports that it has determined that software is not able to recover execution. The PE might use IMPLEMENTATION DEFINED additional syndrome registers to report that software is able to recover execution if software takes action to locate and repair the error using the IMPLEMENTATION DEFINED mechanisms.

I_{RLQVJ} See also:

- PE error state recording in the exception syndrome.
- PE error state classification.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D10.4.1 PE error state recording in the exception syndrome

R_{WPKYM} When an asynchronous SError exception is taken to AArch64 state, the PE records the PE error state in the ESR_ELx exception syndrome register as the applicable one of:

- Uncontainable (UC).
- Unrecoverable state (UEU).
- Recoverable state (UER).
- Restartable state (UEO).
- Corrected (CE).
- Uncategorized error.
- IMPLEMENTATION DEFINED syndrome.

I_{SDDLL} When an asynchronous SError exception is taken to AArch64 state:

- Uncategorized error is recorded by setting ESR_ELx.ISS to zero. This includes setting ESR_ELx.IDS and ESR_ELx.DFSC to zero.
- IMPLEMENTATION DEFINED syndrome is recorded by setting ESR_ELx.IDS to `0b1`. The remainder of the ESR_ELx.ISS syndrome is IMPLEMENTATION DEFINED.

Other values for the PE error state are recorded in ESR_ELx.AET, by setting ESR_ELx.IDS to `0b0` and ESR_ELx.DFSC to the applicable nonzero fault status code, indicating ESR_ELx.AET is valid.

R_{FKHHF} When a synchronous External abort exception is taken, the PE records the PE error state in ESR_ELx.SET as the applicable one of the following:

- Uncontainable (UC).
- Recoverable state (UER).
- Restartable state (UEO).

Other values for the PE error state are not supported by synchronous External abort exceptions.

I_{QVVSM} Table D10-1 summarizes the supported PE error state syndrome values for each type of Error exception.

Table D10-1 Summary of Error exception types and supported PE error state syndrome values

PE error state	External abort	SError exception
Recorded in:	ESR_ELx.SET	ESR_ELx.AET
Uncategorized error	No	Yes (ISS==0)
IMPLEMENTATION DEFINED syndrome	No	Yes (IDS==1)
Uncontainable (UC),	Yes (`0b10`)	Yes (`0b000`)
Unrecoverable state (UEU)	No	Yes (`0b001`)
Recoverable state (UER)	Yes (`0b00`)	Yes (`0b011`)
Restartable state (UEO)	Yes (`0b11`)	Yes (`0b010`)
Deferred (DE)	No	No
Corrected (CE)	No	Yes (`0b110`)

D10.4.2 PE error state classification

I_{CCKWK} The PE determines which PE error state to record based on the following criteria:

- The PE error state syndrome values supported by the type of Error exception being taken. See PE error state recording in the exception syndrome.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The following IMPLEMENTATION SPECIFIC properties and behaviors of the PE on taking the exception:
  - Whether the error has been silently propagated by the PE.
  - Whether the PE determines that software is able to recover execution at the point where the exception is taken.
  - If the PE determines that software can recover execution, whether software needs locate and repair the error before attempting to recover. If software does not locate and repair the error, then attempting to recover execution might cause the Error exception to be generated again.
  - If the PE determines that software cannot recover execution, whether the error is a synchronizable error.
- Whether the implementation elects to record the PE error state as another state. The PE only does this when the criteria for the other, recorded state are met. The conditions under which the PE elects to record the PE error state as another state are IMPLEMENTATION DEFINED.

The recorded PE error state is defined by the rules in this section.

R_{QKZLB} If and only if all of the following are true, then on taking an Error exception the PE error state is recorded as Uncontainable (UC):

- One or more of the following are true:
  - The error has been silently propagated by the PE.
  - The PE determines that software is not able to recover execution from the preferred return address of the exception and the error is not a synchronizable error.
  - The PE determines that software is not able to recover execution from the preferred return address of the exception and the Error exception is taken as a synchronous External abort to AArch64 state. (That is, the type of Error exception does not support reporting the PE error state as Unrecoverable state (UEU).)
  - The implementation has elected to record the PE error state as Uncontainable (UC).
- The implementation has not elected to record the PE error state as IMPLEMENTATION DEFINED syndrome or Uncategorized error, or the type of Error exception does not support reporting the PE error state as IMPLEMENTATION DEFINED syndrome or Uncategorized error.

R_{QGNYD} If and only if all of the following are true, then on taking an Error exception the PE error state is recorded as Unrecoverable state (UEU):

- The error has not been silently propagated by the PE.
- The Error exception is taken as an SError exception.
- One or more of the following are true:
  - The PE determines that software is not able to recover execution from the preferred return address of the exception and the error is a synchronizable error.
  - The implementation has elected to record the PE error state as Unrecoverable state (UEU).
- The implementation has not elected to record the PE error state as Uncontainable (UC), IMPLEMENTATION DEFINED syndrome, or Uncategorized error.

I_{FJCZP} Error synchronization event defines synchronizable error.

R_{JHNVT} If and only if all of the following are true, then on taking an Error exception the PE error state is recorded as Recoverable state (UER):

- The error has not been silently propagated by the PE.
- The PE determines that software is able to recover execution from the preferred return address of the exception.
- One or more of the following are true:
  - The PE determines that software must take action to locate and repair the error to successfully recover execution. This might be because the exception was taken before the error was architecturally consumed by the PE, at the point when the PE was not be able to make correct progress without either consuming the error or otherwise making the state of the PE unrecoverable.
  - The implementation has elected to record the PE error state as Recoverable state (UER).
- The implementation has not elected to record the PE error state as Unrecoverable state (UEU), Uncontainable (UC), IMPLEMENTATION DEFINED syndrome, or Uncategorized error.

R_{MBVCF} If and only if all of the following are true, then on taking an Error exception the PE error state is recorded as Restartable state (UEO):

- The error has not been silently propagated by the PE.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The PE determines that software can recover execution from the preferred return address of the exception without the need for software to take action to locate and repair the error first.
- One or more of the following are true:
  - The error is an uncorrected error. This includes a deferred error.
  - The error is a corrected error and the Error exception is not taken as an SError exception taken to AArch64 state.
  - The implementation has elected to record the PE error state as Restartable state (UEO).
- The implementation has not elected to record the PE error state as any of Recoverable state (UER), Unrecoverable state (UEU), Uncontainable (UC), IMPLEMENTATION DEFINED syndrome, or Uncategorized error.

R_{LFXRD} If and only if all of the following are true, then on taking an Error exception the PE error state is recorded as Corrected (CE):

- The error has been corrected and not silently propagated by the PE.
- The Error exception is taken as an SError exception taken to AArch64 state.
- Software can recover execution from the preferred return address of the exception. Because the error has been corrected, software does not need to take action to locate and repair the error.
- The implementation has not elected to record the PE error state as any other type.

R_{NZYRP} If and only if all the following are true, then on taking an Error exception the PE error state is recorded as an Uncategorized error:

- The Error exception is taken as an asynchronous SError exception taken to AArch64 state.
- The implementation has elected to record the PE error state as an Uncategorized error.

R_{VHWHD} If and only if all the following are true, then on taking an Error exception the PE error state is recorded as an IMPLEMENTATION DEFINED syndrome:

- The Error exception is taken as an asynchronous SError exception taken to AArch64 state.
- The implementation has elected to record the PE error state as an IMPLEMENTATION DEFINED syndrome.

I_{SRPJD} The IMPLEMENTATION DEFINED syndrome type might provide additional IMPLEMENTATION DEFINED syndrome recorded in the exception syndrome register. Software might be able to determine the state of the PE from this syndrome, or other IMPLEMENTATION DEFINED syndrome registers.

I_{WLZRP} Uncategorized error and IMPLEMENTATION DEFINED syndrome are defined for backward compatibility with previous versions of the architecture. Arm does not recommend use of these PE error state values in new implementations that include other RAS features.

I_{VKMZB} The PE error states are summarized by Figure D10-1. Figure D10-1 assumes the type of Error exception supports the resulting PE error state, never elects to record an error as a different PE error state when permitted, and does not show Uncategorized error or IMPLEMENTATION DEFINED syndrome.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

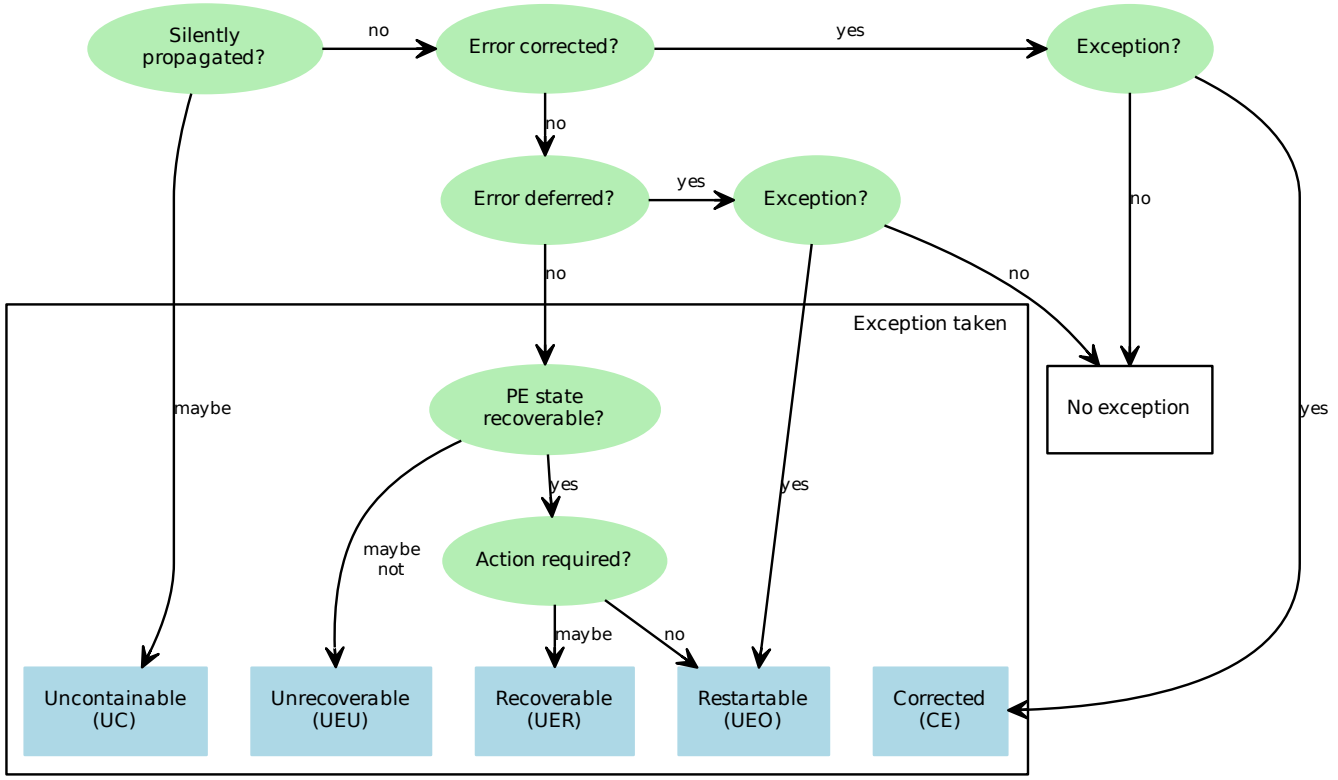


Figure D10-1 PE error states

I_{ZQRGL} If the PE error state reports that software can recover execution, or that software isolation might be possible because the error is a synchronizable error, then this does not necessarily mean that the error can be recovered from because the error in the system might be one which does not allow software to recover the operation. Rather, software *might* be able to recover if it can repair the error and continue.

Example D10-12

A component detects an error when accessed by a PE, and records in a RAS System Architecture node that the error is uncontainable at the component, meaning the system has to be shut down to avoid catastrophic failure. The component signals the error with an in-band error response to a PE, which does not report the severity of the error. The recorded PE error state refers only to the PE, not the system error state.

I_{WBVYC} If the in-band error response can signal the severity of the error to the PE, then the PE might use this information to elect to report the PE error state as the severity of error reported to it, if permitted by the preceding rules.
However, this is not required, and software must not rely on this behavior and should determine from the system whether the error is recoverable at the system level.

Example D10-13

A processor cache detects an uncontainable tag RAM error, and the PE reports the PE error state as Uncontainable (UC), even though the state of the PE itself is Recoverable state (UER). R_{QKZLB} allows this.

I_{TXFMY} An SError exception that is masked when it is generated remains pending until taken later when no longer masked.
In these cases, the PE error state is also unlikely to be *Restartable* or *Recoverable*.

I_{TGPQZ} See also:

- PE error propagation.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Error synchronization event.

D10.4.2.1 Using the PE error state classification

S_{XSKNS} When the PE error state is recorded as Uncontainable (UC):

- The error handling software must assume that either:
  - The error has been silently propagated by the PE.
  - Software is not able to recover execution from the preferred return address of the exception and the error was not a synchronizable error.
- If the error handling software cannot isolate the error to an application or VM, or both, by other means, then the system must be shut down by software to avoid catastrophic failure.

S_{HYWFL} When the PE error state is recorded as Unrecoverable state (UEU):

- The error handling software can assume the error has not been silently propagated by the PE.
- The error handler cannot safely recover execution from the preferred return address of the exception, even if it takes action to locate and repair the error. The state of the affected software, or both, is unrecoverable. However, if the software includes Error synchronization events, the error handler can use the properties of the Error synchronization event to determine which software is affected by the error.
- The affected software cannot continue and must be isolated by the error handling software.

S_{LSFYM} When the PE error state is recorded as Recoverable state (UER):

- The uncorrected error might remain latent in the system.
- If the error handling software takes action to locate and repair the uncorrected error, then the error handler can safely recover execution from the preferred return address of the exception. Otherwise on restart of the affected software the PE might attempt to consume the error again, causing a further Error exception. If the error handler cannot locate and repair the error, then the affected software must be isolated by the error handling software.

S_{GLPZY} When the PE error state is recorded as Restartable state (UEO):

- The error might remain latent in the system.
- The error handling software might take action to locate and repair the error before it is consumed. However, the affected software can be safely restarted by the error handler without software taking any action to locate and repair the error.

For example, the error was signaled when the PE speculatively accessed corrupt data.

S_{GRZQS} When the PE error state is recorded as IMPLEMENTATION DEFINED syndrome or Uncategorized error, if the error handling software is not able to determine the actual state of the PE and memory, it should treat IMPLEMENTATION DEFINED syndrome and Uncategorized error as Uncontainable (UC).

D10.4.3 Multiple SError exceptions

I_{CPJLW} Multiple physical and/or virtual SError exception conditions might be pending together. The architecture does not define relative priorities for asynchronous exceptions.

R_{DHKQZ} If multiple physical and/or virtual SError exception conditions are pending, then it is IMPLEMENTATION DEFINED whether the multiple pending SError exception conditions are taken as a single SError exception.

R_{JBQSC} On taking an SError exception for more than one SError condition:

- If the exception is taken and one or more pending SError conditions would be reported as IMPLEMENTATION DEFINED syndrome or Uncategorized error, then the syndrome recorded in ESR_ELx.ISS is IMPLEMENTATION DEFINED.
- Otherwise, the recorded PE error state applies recorded by combined effect of the errors.

I_{GNHXJ} Any pending SError conditions that are not taken with other SError exceptions as a single SError exception remains pending after the SError exception is taken.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D10.4.4 Target Exception level for External abort and SError exceptions taken to AArch64 state

D_{FWSVH} This section is included for completeness. It repeats the definitions from this Manual and so is *non-normative*.

I_{NRZXZ} The default target Exception level for synchronous External abort exceptions is:

- EL1, if taken from EL0 or EL1.
- EL2, if taken from EL2.

The default target Exception level for SError exceptions is EL1.

However:

- If HCR_EL2.AMO is `0b1` or HCR_EL2.TGE is `0b1`, then the target Exception level for SError exceptions is EL2.
- If HCR_EL2.TEA is `0b1` or HCR_EL2.TGE is `0b1`, then the target Exception level for synchronous External abort exceptions taken from EL0 and EL1 is EL2.

When executing at a higher Exception level than the target Exception level for SError exceptions, SError exceptions are implicitly masked and not taken.

I_{BPGXV} See also The AArch64 System Level Programmers' Model.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D10.5 Error synchronization event

R_{GRJVN} An Error synchronization event is generated by any of the following:

- Executing an ESB instruction.
- When FEAT_IESB is implemented, and one of the following is true, taking an exception to an Exception level, ELx, the appropriate SCTLR_ELx.IESB bit is 0b1.
  In Debug state this also applies to executing a DCPSx instruction to ELx.
- When FEAT_IESB is implemented, and one of the following is true, executing an exception return instruction at an Exception level, ELx, the appropriate SCTLR_ELx.IESB bit is 0b1.
  In Debug state this also applies to executing a DRET instruction at ELx.

I_{PMGVM} In addition to generating an Error synchronization event, the ESB instruction might additionally record and then clear a masked pending asynchronous SError exception. This is also referred to as deferring the pending asynchronous SError exception.

R_{WPPWC} A synchronizable error is one of a subset of errors specific to the implementation that are synchronized by Error synchronization events, generated by an instruction executed on the same PE as the Error synchronization events.

Note: Synchronizable error must not be confused with a synchronous error exception. An Error exception for a synchronizable error might be taken asynchronously.

R_{BZPTV} If the Error exception for a synchronizable error is taken in program order after the Error synchronization event completes, and either physical SError exceptions are unmasked when the Error synchronization event occurs or the Error exception is taken synchronously, then all of the following are true:

- The instruction that generated the synchronizable error is in program order after the Error synchronization event.
- On completion of the Error synchronization event, the PE state and memory system state are consistent with the PE having executed all instructions in program order before the Error synchronization event.

R_{ZRHDD} If the Error exception for a synchronizable error is taken asynchronously as an SError exception, physical SError exceptions are masked when the Error synchronization event occurs, and the SError exception is not pending when the Error synchronization event completes, then all of the following are true:

- The instruction that generated the synchronizable error is in program order after the Error synchronization event.
- On completion of the Error synchronization event, the PE state and memory system state are consistent with the PE having executed all instructions in program order before the Error synchronization event.

The SError exception is not pending when the Error synchronization event completes if a subsequent read of ISR_EL1.A returns 0b0.

R_{YCJTN} If the Error exception for a synchronizable error is taken asynchronously as an SError exception, the Error synchronization event is generated by an ESB instruction executed when physical SError exceptions are masked, and the ESB instruction does not set DISR_EL1.A to 0b1, then all of the following are true:

- The instruction that generated the synchronizable error is in program order after the ESB.
- On completion of the ESB, the PE state and memory system state are consistent with the PE having executed all instructions in program order before the ESB.

R_{NFKMQ} Taken in program order after the Error synchronization event completes means:

- For an Error synchronization event generated by an ESB instruction, the exception is taken in program order after the instruction.
- For an Error synchronization event generated by an exception return instruction when FEAT_IESB implemented, the exception is taken in program order after the instruction.
- For an Error synchronization event generated by an exception entry when FEAT_IESB is implemented, one of the following is true:
  - The exception is taken in program order strictly after the first instruction of the exception handler at the exception vector address.
  - The exception is taken from the first instruction of the exception handler at the exception vector address and the ESR_ELx.IESB syndrome bit is recorded as 0b0.

I_{QZSHG} The definition of synchronizable error means that if the error that is a synchronizable error is generated by an instruction in program order before the Error synchronization event, then either the Error exception is taken before the Error synchronization event, or on executing the Error synchronization event the following apply:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If physical SError exceptions are unmasked or the Error exception is taken synchronously, then the Error synchronization event ensures that the Error exception is not taken in program order after the Error synchronization event. This allows isolation of the software affected by the error.
- If physical SError exceptions are masked and the Error exception is taken asynchronously, then:
  - If the Error synchronization event was generated by an ESB, then the error is recorded in DISR_EL1. Software can use the PE error state recorded in DISR_EL1 to determine what recovery is possible.
  - Otherwise, the Error exception is pending when the Error synchronization event completes.

The SError exception might have been pending before or made pending by the Error synchronization event.

The definition does not mean that if the error is generated by a instruction in program order after the Error synchronization event, then the Error exception will only be taken after the Error synchronization event. The Error exception might be taken before the Error synchronization event, if the PE speculated past the Error synchronization event and speculatively executed the instruction that generated the error. This might cause software to generate a false failure. Error synchronization events are not speculation barriers.

I_{SQCFG} The criteria for the PE error state mean that if the PE reports the PE error state as one of the following, then the error must be either explicitly or implicitly a synchronizable error:

- Unrecoverable state (UEU).
- Recoverable state (UER).
- Restartable state (UEO).

This is because whether the error is a synchronizable error is a criterion for Unrecoverable state (UEU), and the criteria for Recoverable state (UER) and Restartable state (UEO) satisfy the definition of synchronizable error.

For other physical errors:

- An error that has been silently propagated by the PE and is not reported as either IMPLEMENTATION DEFINED syndrome or Uncategorized error must be reported as Uncontainable (UC) and is not containable even if the error is a synchronizable error. Software must assume the error has been silently propagated even if the error is a synchronizable error.
- It is IMPLEMENTATION SPECIFIC whether an error reported with an ESR_ELx.ISS syndrome that is IMPLEMENTATION DEFINED syndrome or Uncategorized error is a synchronizable error.
- The following errors have not been consumed by the PE:
  - A *Deferred error*.
  - A *Corrected error*.
  - An Error exception from a read by hardware speculation that does not corrupt the state of the PE.

  Software can recover execution from these errors regardless of whether the error is a synchronizable error.
- An implementation might have other IMPLEMENTATION DEFINED Error exception and other sources of SError exception, see R_{FNVVJ}. If an IMPLEMENTATION DEFINED SError exception is generated by a level-sensitive interrupt signal, then the SError exception cannot be a synchronizable error.

I_{VFFYW} An Error synchronization event might operate as follows:

1. The PE ensures that any synchronizable error generated by an instruction in program order before the Error synchronization event has caused a physical SError exception to become pending.
2. If a physical SError exception is pending for a synchronizable error and generated by an instruction in program order before the Error synchronization event, and physical SError exceptions are not masked at the current Exception level, then the physical SError exception is taken before completion of the Error synchronization event. The SError exception might have been made pending by the Error synchronization event, or might have been pending before the Error synchronization event.

R_{NPPGJ} If an SError exception for a synchronizable error is pending after completing the Error synchronization event generated by an ESB instruction, and physical SError exceptions are masked at the current Exception level, then the ESB instruction performs the following steps:

1. The pending physical SError exception is recorded in DISR_EL1. This includes the PE error state that the pending Error exception would record if taken.
2. The DISR_EL1.A bit bit is set to `0b1`.
3. The pending state of the physical SError exception is cleared.

The SError exception might have been made pending by the Error synchronization event, or might have been pending before the Error synchronization event.

R_{BLRTM} The criteria for ESB recording the PE error state in DISR_EL1 are the same as for that for recording the PE error state in ESR_ELx when an SError exception taken to the current execution state.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{KNWBN} If an SError exception is taken as part of an Error synchronization event generated by an ESB instruction, then the ESB instruction address is the *preferred return address* of the exception.

For the definition of the *preferred return address* for an exception, see Preferred return address.

R_{SFHDS} On executing an ESB instruction when SError exceptions are masked, any pending SError exception generated by an error that is not a synchronizable error:

- Remains pending after completion of the Error synchronization event.
- Does not update DISR_EL1.

I_{CQKXL} The Error recovery interrupt, Fault handling interrupt, and Critical Error interrupt interrupts described by RAS System Architecture are asynchronous interrupts, and so are not synchronizable errors.

I_{BBGXN} If multiple SError exception conditions are pending, then an Error synchronization event synchronizes all errors that are synchronizable errors.

S_{VFHGT} Software must be aware that an SError exception taken at an Error synchronization event or recorded in the DISR_EL1 register by an ESB instruction might have been generated by hardware speculation of an instruction in program order after the Error synchronization event.

D10.5.1 ESB and virtual SError exceptions

R_{TFVYW} If VSESR_EL2 is writable, then an ESB instruction executed at EL0 or EL1 synchronizes a pending virtual SError exception.

In these cases, a virtual SError exception is pending, and the following occur when an ESB instruction is executed at EL0 or EL1:

- If the virtual SError exception is unmasked at the current Exception level, then the exception is taken before the completion of the ESB instruction.
- If the virtual SError exception is masked at the current Exception level, then all the following occur:
  - HCR_EL2.VSE cleared to 0b0.
  - The virtual SError exception syndrome from VSESR_EL2 is recorded in VDISR_EL2. See R_{HDCTW} and R_{FLYGZ}.
  - VDISR_EL2.A is set to 0b1 to indicate the SError exception was pending prior to the execution of the ESB instruction.

R_{GXHYX} If all of the following are true, then it is IMPLEMENTATION DEFINED whether or not an ESB instruction executed at EL0 or EL1 synchronizes a pending virtual SError exception:

- EL2 is implemented and enabled in the current Security state.
- EL2 is using AArch64, HCR_EL2.TGE is 0b0, HCR_EL2.VSE is 0b1 and either:
  - HCR_EL2.AMO is 0b1.
- VSESR_EL2 is implemented as RAZ/WI.

In these cases, a virtual SError exception is pending. If the IMPLEMENTATION DEFINED choice is that the ESB instruction synchronizes a pending virtual SError exception, then the following occur when an ESB instruction is executed at EL0 or EL1:

- If the virtual SError exception is unmasked at the current Exception level, then the exception is taken before the completion of the ESB instruction.
- If the virtual SError exception is masked at the current Exception level, then all the following occur:
  - HCR_EL2.VSE cleared to 0b0.
  - The virtual SError exception syndrome in VDISR_EL2 is set to zero.
  - VDISR_EL2.A is set to 0b1 to indicate the virtual SError exception was pending prior to the execution of the ESB instruction.

If the IMPLEMENTATION DEFINED choice is that the ESB instruction does not synchronize a pending virtual SError exception, then an ESB instruction executed at EL0 or EL1 ignores the pending virtual SError exception and the virtual SError exception stays pending.

See also Syndrome registers.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{VKQGL} If both a virtual SError exception and a physical SError exception are pending as described in R_{LLLVR} and Error synchronization event, then the physical SError exception is not masked at the current Exception level, because this is not possible when a virtual SError exception is pending. This means that:

- If the virtual SError exception is masked at the current Exception level, then the physical SError exception will be taken before the `ESB` instruction completes and HCR_EL2.VSE and VDISR_EL2 are unchanged.
- If the virtual SError exception is not masked at the current Exception level, then it is IMPLEMENTATION DEFINED which of the virtual SError or physical SError exceptions is taken before the `ESB` instruction completes. If the virtual SError exception is taken and, after taking the exception to EL1, the physical SError exception is still pending, then the physical SError exception is still not masked and will be taken.

D10.5.2 Extension for synchronization at exception entry and return

D_{DPSJR} The rules in this section apply when FEAT_IESB is implemented.

I_{WDSBL} An implicit Error synchronization event has no effect on DISR_EL1 or VDISR_EL2.

D10.5.2.1 Synchronization on exception entry

R_{RNZWY} For each value of ELx in EL1 or EL2, if all of the following are true, then each exception that is taken to ELx generates an Error synchronization event:

- The Effective value of SCTLR_ELx.IESB is `0b1`.

R_{RPBWR} For each value of ELx in EL1 or EL2 if all of the following are true, then executing a `DCPSx` instruction generates an Error synchronization event:

- The PE is in Debug state.
- The Effective value of SCTLR_ELx.IESB is `0b1`.

R_{JQSKQ} If an SError exception is taken to the Exception level ELy as a result of the Error synchronization event generated on exception entry by the FEAT_IESB mechanism, then all the following occur:

- The PE sets the ESR_ELy.IESB bit in the SError exception syndrome to `0b1`.
- The preferred return address for the SError exception is the exception vector address for the original exception.

Note: ELy might be the same Exception level as ELx.

I_{FWZHV} If SError exceptions are masked at ELx, then any SError exception made pending by the Error synchronization event stays pending.

I_{MMVJW} The prioritization of asynchronous exceptions is IMPLEMENTATION DEFINED. This means that an implementation might choose to behave as if the SError exception made pending by the Error synchronization event was taken before the implicit Error synchronization event, if the SError exception was not masked, taking the SError exception in place of the original exception, unless the original exception is also an SError exception.

In this case, ESR_ELy.IESB is set to `0b0` and the reported PE error state correctly indicates, for instance, whether software can recover execution from the preferred return address for the SError exception in ELR_ELy.

D10.5.2.2 Synchronization on exception return

R_{SKRCR} For each value of ELx in EL1 or EL2 if all of the following are true, then executing an exception return instruction at ELx generates an Error synchronization event:

- The instruction does not generate any exception.
- The Effective value of SCTLR_ELx.IESB is `0b1`.

I_{GPPXQ} On an illegal return event, the exception return instruction sets PSTATE.IL to `0b1`, which causes the next instruction to generate an Illegal State exception. The exception return instruction does not generate the exception.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R_{CVPDN} For each value of ELx in EL1 or EL2 if all of the following are true, then executing an `DRPS` instruction at ELx generates an Error synchronization event:

- The PE is in Debug state and the instruction does not generate any exception.
- The Effective value of SCTLR_ELx.IESB is `0b1`.

R_{GXQYD} Any SError exception taken as part of the Error synchronization event terminates execution of the instruction.

R_{LPKVM} If an SError exception is taken to an Exception level, ELy, as a result of the Error synchronization event generated on exception return by the FEAT_IESB mechanism, then all the following occur:

- The PE sets the ESR_ELy.IESB bit in the SError exception syndrome to an IMPLEMENTATION DEFINED choice of `0b0` or `0b1`.
- The preferred return address for the SError exception is the address of the `ERET` instruction.

I_{JZHDB} If SError exceptions are masked at ELx, then any SError exception made pending by the Error synchronization event stays pending.

D10.5.3 Error synchronization barriers in a minimal implementation

I_{HQRLS} Error synchronization events and the `ESB` instruction can be implemented as no-ops if either there are no sources of SError exceptions, or all SError exceptions are reported as Uncategorized error and are not synchronizable errors.
This allows for a very low cost implementation of the RAS Extension.
Such an implementation might include FEAT_IESB, meaning the SCTLR_ELx.IESB controls have no effect.

I_{HSHJG} See also Syndrome registers.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D10.6 Virtual SError exceptions and delegated SError exceptions

I_{LSSCN} EL2 provides a virtual SError exception.

For more information, see:

- Virtual interrupts
- Virtual exceptions when an implementation includes EL2.

I_{ZXMPL} FEAT_RAS provides:

- Mechanisms to allow a hypervisor to specify the syndrome value reported to a guest Operating System on taking a virtual SError exception.
- Mechanisms to allow firmware to specify the syndrome value reported to a hypervisor or guest Operating System on taking a delegated SError exception.
- Support for EL0 or EL1 to isolate a virtual SError exception and for EL0, EL1, or EL2 to isolate a delegated SError exception as if it was a physical SError exception. See ESB and virtual SError exceptions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D10.7 Error records in the PE

I_{KMMPK} A component that records detected errors is called a node by the RAS System Architecture. Each node implements one or more Error records.

R_{VNLPC} It is IMPLEMENTATION DEFINED whether the processor that implements a PE implements any nodes.

R_{XKDRX} A PE implementing the RAS Extension might implement the System register interface to nodes.

I_{SCVSB} The System register interface to nodes is not restricted to accessing only PE nodes.

I_{ZRKKQ} When an error is recorded by a PE node, one or more of the following might be generated, according to the configuration of the node:

- A Fault handling interrupt.
- An Error recovery interrupt.
- A Critical Error interrupt.
- An in-band error response.

D10.7.1 Error record System register view

I_{SLVDW} If the System register interface to a node is implemented, then software can access the Error records of the node using Error record System registers.

R_{BYLZQ} The number of Error records that can be accessed using the System registers is IMPLEMENTATION DEFINED, and might be zero. The ERRIDR_EL1 registers indicate the highest numbered index of the Error records that can be accessed using System registers, plus one.

I_{NWBNQ} The AArch64 Error record System registers are those registers with an ERX*_EL1 mnemonic.

I_{BJNDF} The following table describes the AArch64 Error record System registers.
The descriptions in this section apply whether the Error record is accessed through the indirection mechanism, as described in Table D10-2, or as memory-mapped registers, as described in RAS registers summary.

Table D10-2 Using AArch64 System registers, System register map

Use	To Access	Access	Description
ERXADDR_EL1	ERR<n>ADDR	RW	Error Record <*n*> Address Register
ERXCTLR_EL1	ERR<n>CTLR	RW	Error Record <*n*> Control Register
ERXFR_EL1	ERR<n>FR	RO	Error Record <*n*> Feature Register
ERXMISC0_EL1	ERR<n>MISC0	RW	Error Record <*n*> Miscellaneous Register 0
ERXMISC1_EL1	ERR<n>MISC1	RW	Error Record <*n*> Miscellaneous Register 1
ERXMISC2_EL1[a]	ERR<n>MISC2	RW	Error Record <*n*> Miscellaneous Register 2
ERXMISC3_EL1[a]	ERR<n>MISC3	RW	Error Record <*n*> Miscellaneous Register 3
ERXPFGCDN_EL1	ERR<n>PFGCDN	RW	Error Record <*n*> Pseudo-fault Generation Countdown Register
ERXPFGCTL_EL1	ERR<n>PFGCTL	RW	Error Record <*n*> Pseudo-fault Generation Control Register
ERXPFGF_EL1	ERR<n>PFGF	RO	Error Record <*n*> Pseudo-fault Generation Feature Register
ERXSTATUS_EL1	ERR<n>STATUS	RW	Error Record <*n*> Primary Status Register

a. If FEAT_RASv1p1 is implemented.

R_{ZBCFZ} If FEAT_RASv1p1 is implemented, then all Error records accessible through System registers implement RAS System Architecture version 1.1.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

S_{VBBNY} To access an Error record, software:

1. Sets the error selection register, ERRSELR_EL1.SEL or ERRSELR.SEL, to the index of the record being accessed.
2. Accesses the Error record using the ERX*_EL1 or ERX* System registers, as described in I_{BJNDF}.

I_{WKXSB} The Error records accessed through the System registers might be accessible only to the PE associated with those System registers, or they might be shared and therefore accessible to other PEs through either System registers or as a memory-mapped component.

I_{JMVVD} Direct reads of the System registers, including Error record System registers, can occur speculatively and out-of-order relative to other instructions executed on the same PE.

R_{WFPWF} Direct reads and writes of the Error records through the ERX*_EL1 AArch64 System registers, as described in I_{BJNDF}, are indirect reads of ERRSELR_EL1.

D10.7.1.1 Syndrome registers

I_{RGMHN} ESR_ELx is the exception syndrome registers. The PE records syndrome information in an exception syndrome register on taking a physical SError exception or synchronous External abort exception. ESR_ELx is also used by other exceptions.
DISR_EL1 is the deferred error syndrome registers. The PE records syndrome information in a deferred error syndrome register on deferring a physical SError exception.
The PE also records a virtual syndrome value in ESR_EL1 or DISR_EL1 on taking or deferring a virtual SError exception. The virtual syndrome value is provided by software in a corresponding virtual error syndrome register, VSESR_EL2 or VDISR_EL2 respectively.
The PE also records a delegated syndrome value in ESR_EL2, ESR_EL1, VDISR_EL2, or DISR_EL1 on taking or deferring a delegated SError exception.

R_{SLNMV} For a given implementation:

- If ESB never synchronizes any errors, then DISR_EL1.A might be RES0.
- The deferred and virtual syndrome registers are capable of storing any syndrome value that might be recorded by the PE in an exception syndrome register on taking a physical SError exception or synchronous External abort exception.
- If any of ESR_ELx[24:0] is not used and always set to zero by the PE on taking a physical SError exception or synchronous External abort exception, then that bit can be RES0 in that exception syndrome register.
- A bit that is not used and always set to zero or always set to one by the PE on taking a physical SError exception is permitted to be RES0 or RES1 respectively in the corresponding deferred and virtual error syndrome registers. See Table D10-3.

In Table D10-3, the deferred or virtual error syndrome register bit described in the left-hand column is permitted to be RES0 or RES1 only if the corresponding bit is always set to zero or always set to one (respectively) on taking a physical SError exception in all of the implemented exception syndrome registers listed in the other columns marked *Yes* on that row. Otherwise, the bit is read/write.

Table D10-3 Permitted relaxations for bits in deferred and virtual error syndrome registers

Bit is permitted to be res0 or res1	ESR_ELx[*n*] *n* ∈ [24:0]
VSESR_EL2[*n*]	Yes
VDISR_EL2[*n*]	Yes
DISR_EL1[*n*]	Yes

I_{NQNWV} R_{SLNMV} means that VSESR_EL2 can be implemented as RAZ/WI when ESR_ELx[24:0] are always set to zero by the PE on taking a physical SError exception or synchronous External abort exception. When this is the case, the PE error state is always reported as Uncategorized error when a physical SError exception is taken.
I_{GQQCK}, in Error synchronization barriers in a minimal implementation then further allows ESB to be executed as a NOP, meaning DISR_EL1 and VDISR_EL2 can also be implemented as RAZ/WI.
This allows for a very low-cost implementation of the RAS Extension.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter D11
AArch64 System Register Encoding

This chapter describes the AArch64 System register encoding space. It contains the following sections:

- The System register encoding space.
- Moves to and from debug and trace System registers.
- Moves to and from non-debug System registers, Special-purpose registers.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D11.1 The System register encoding space

The A64 instruction set includes instructions that access the System register encoding space. These instructions provide:

- Access to *System registers*, including the debug registers, that provide system control, and system status information.
- Access to Special-purpose registers such as SPSR_ELx, ELR_ELx, and the equivalent fields of the Process State.
- The cache and TLB maintenance instructions and address translation instructions.
- Barriers and the `CLREX` instruction.
- Architectural hint instructions.

This section describes the parts of the System register encoding space that provides access to the System registers described in AArch64 System Register Descriptions.

Note

- See Fixed values in AArch64 instruction and System register descriptions for information about abbreviations used in the System instruction descriptions.

Principles of the System instruction class encoding describes some general properties of these encodings. System instruction class encoding overview then describes the top-level encoding of these instructions, identifying that:

- Entries in the encoding space are characterized by the parameter set {`op0`, `op1`, `CRn`, `CRm`, `op2`}.
- `op0` is the most significant parameter for determining allocations in this space.

Much of this encoding space is used for System instructions, as described in The A64 System Instruction Class. This chapter describes only the part of the encoding space that is used for System registers, in the following sections:

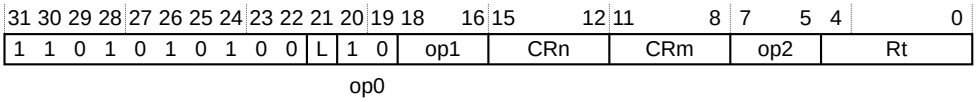
- Moves to and from debug and trace System registers.
- Moves to and from non-debug System registers, Special-purpose registers.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D11.2 Moves to and from debug and trace System registers

The instructions that move data to and from the debug, execution environment, and trace System registers are encoded with `op0== 0b10`. This means the encoding of these instructions is:



Note

- The section describes the use of all of the `op0==0b10` region of the System register encoding space.

The value of `op1` provides the next level of decode of these instructions, as follows:

`op1` == {`0b000`, `0b011`, `0b100`, `0b101`, `0b110`, `0b111`}

Debug, PMU, and System PMU registers.

Note

The standard encoding of debug registers is `op0` == `0b10`, `op1` == {0, 3, 4}. The registers in the `op0` == `0b11` encoding space that are classified as debug registers are DLR_EL0, DSPSR_EL0, MDCR_EL2,. See Instructions for accessing non-debug System registers for the encodings of these registers.

`op1` == {`0b001`} Trace and BRBE registers.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D11.2.1 Instructions for accessing debug System registers

The instructions for accessing debug System registers are:

```
MSR <System_register>, Xt    ; Write to System register
MRS Xt, <System_register>    ; Read from System register
```

Where `<System_register>` is the register name, for example MDCCSR_EL0.

This section includes only the System register access encodings for which both:

- `op0` is `0b10`.
- The value of `op1` is one of {`0b000`, `0b001`, `0b011`, `0b100`, `0b101`, `0b110`, `0b111`}.

Table D11-1 shows the mapping of the System register encodings for debug System register access.

Table D11-1 Instruction encodings for debug System register access in the (`op0==0b10`) encoding space

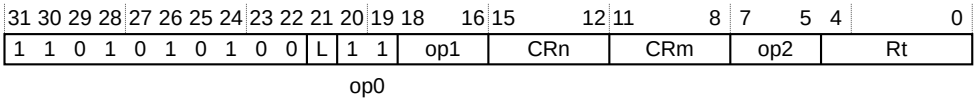
op1	CRn	CRm	op2	Access	Mnemonic	Accesses
0b000	0b0000	0b0000	0b010	RW	OSDTRRX_EL1	OSDTRRX_EL1
0b000	0b0000	0b0010	0b000	RW	MDCCINT_EL1	MDCCINT_EL1
0b000	0b0000	0b0010	0b010	RW	MDSCR_EL1	MDSCR_EL1
0b000	0b0000	0b0011	0b010	RW	OSDTRTX_EL1	OSDTRTX_EL1
0b000	0b0000	0b0110	0b010	RW	OSECCR_EL1	OSECCR_EL1
0b000	0b0000	m[3:0]	0b100	RW	DBGBVR<m>_EL1	DBGBVR<n>_EL1
0b000	0b0000	m[3:0]	0b101	RW	DBGBCR<m>_EL1	DBGBCR<n>_EL1
0b000	0b0000	m[3:0]	0b110	RW	DBGWVR<m>_EL1	DBGWVR<n>_EL1
0b000	0b0000	m[3:0]	0b111	RW	DBGWCR<m>_EL1	DBGWCR<n>_EL1
0b000	0b0001	0b0000	0b000	RO	MDRAR_EL1	MDRAR_EL1
0b000	0b0001	0b0000	0b100	WO	OSLAR_EL1	OSLAR_EL1
0b000	0b0001	0b0001	0b100	RO	OSLSR_EL1	OSLSR_EL1
0b000	0b0001	0b0011	0b100	RW	OSDLR_EL1	OSDLR_EL1
0b000	0b0001	0b0100	0b100	RW	DBGPRCR_EL1	DBGPRCR_EL1
0b000	0b0111	0b1000	0b110	RW	DBGCLAIMSET_EL1	DBGCLAIMSET_EL1
0b000	0b0111	0b1001	0b110	RW	DBGCLAIMCLR_EL1	DBGCLAIMCLR_EL1
0b000	0b0111	0b1110	0b110	RO	DBGAUTHSTATUS_EL1	DBGAUTHSTATUS_EL1
0b011	0b0000	0b0001	0b000	RO	MDCCSR_EL0	MDCCSR_EL0
0b011	0b0000	0b0100	0b000	RW	DBGDTR_EL0	DBGDTR_EL0
0b011	0b0000	0b0101	0b000	RO	DBGDTRRX_EL0	DBGDTRRX_EL0
0b011	0b0000	0b0101	0b000	WO	DBGDTRTX_EL0	DBGDTRTX_EL0

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D11.3 Moves to and from non-debug System registers, Special-purpose registers

The instructions that move data to and from non-debug System registers are encoded with `op0== 0b11`, except that some of this encoding space is reserved for IMPLEMENTATION DEFINED functionality. The encoding of these instructions is:



The value of `CRn` provides the next level of decode of these instructions, as follows:

`CRn` == {`0b0000`, `0b0001`, `0b0010`, `0b0011`, `0b0101`, `0b0110`, `0b0111`, `0b1001`, `0b1010`, `0b1100`, `0b1101`, `0b1110`}

See Instructions for accessing non-debug System registers.

`CRn` == {`0b0100`}

See Instructions for accessing Special-purpose registers.

`CRn`=={`0b1011`, `0b1111`}

See Reserved encodings for IMPLEMENTATION DEFINED registers.

D11.3.1 Instructions for accessing non-debug System registers

The A64 instructions for accessing System registers are:

```
MSR <System_register>, Xt     ; Write to 64-bit System register
MSRR <System_register>, Xt    ; Write to 128-bit System register
MRS Xt, <System_register>     ; Read from 64-bit System register
MRRS Xt, <System_register>    ; Read from 128-bit System register
```

Where `<System_register>` is the register name, for example MIDR_EL1.

This section includes only the System register access encodings for which both:

- `op0` is `0b11`.
- The value of `CRn` is one of {`0b0000`, `0b0001`, `0b0010`, `0b0011`, `0b0100`, `0b0101`, `0b0110`, `0b0111`, `0b1001`, `0b1010`, `0b1100`, `0b1101`, `0b1110`}.

Direct accesses to some regions of the system register encoding space have a defined behavior even if no register is currently allocated or implemented for the encodings in those regions. The Feature ID space is the System register space in AArch64 with `op0==3, op1=={0, 1, 3}, CRn==0, CRm=={0-7}, op2=={0-7}`. The properties for this space are that:

- For unused encodings in the part of the space that has `op1` == `0` and `CRm` == `{2-7}`, the following apply:
  - It is IMPLEMENTATION DEFINED whether direct reads can be trapped by HCR_EL2.TID3.
  - Direct reads that are not trapped behave as Reserved, RES0 to ensure correct behavior if the encodings are used for ID registers in the future.
- For unused encodings in the part of the space that has either `op1` == `0` and `CRm` == `0`, or `op1` == `{1, 3}`, the following apply:
  - Direct reads are UNDEFINED and therefore generate exceptions reported with EC syndrome value `0x00`.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

- While this group is described as accessing the non-debug System registers, its correct characterization is by the {op0, CRn} values given in this subsection, and the group includes the debug registers DLR_EL0, DSPSR_EL0, MDCR_EL2, MDCR_EL3, and SDER32_EL3 and MDCR_EL3, which are described in Debug registers. These registers are exceptions to the standard encoding of debug registers, which has op0 == 0b10, see Instructions for accessing debug System registers.
- All unused encodings in the range Op0 == 3, op1 == 0, CRn == 0, CRm == {2-7}, op2 == {0-7} are defined to be accessible as Reserved, RAZ to ensure correct behavior if the encodings are used for ID registers in the future.

The instruction encoding for these accesses is:

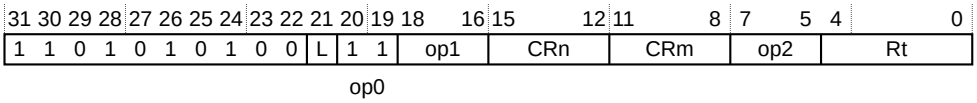


Table D11-2 shows the encodings of the register access instructions.

See the register descriptions for information about the control that determines whether these accesses are permitted.

Table D11-2 Instruction encodings for non-debug System register access in the (op0==0b11) encoding space

op1	CRn	CRm	op2	Access	Mnemonic	Accesses
0b000	0b0000	0b0000	0b000	RO	MIDR_EL1	MIDR_EL1 VPIDR_EL2
0b000	0b0000	0b0000	0b100	RO	MPUIR_EL1	MPUIR_EL1
0b000	0b0000	0b0000	0b101	RO	MPIDR_EL1	MPIDR_EL1 VMPIDR_EL2
0b000	0b0000	0b0000	0b110	RO	REVIDR_EL1	REVIDR_EL1
0b000	0b0000	0b0001	0b000	RO	ID_PFR0_EL1	ID_PFR0_EL1
0b000	0b0000	0b0001	0b001	RO	ID_PFR1_EL1	ID_PFR1_EL1
0b000	0b0000	0b0001	0b010	RO	ID_DFR0_EL1	ID_DFR0_EL1
0b000	0b0000	0b0001	0b011	RO	ID_AFR0_EL1	ID_AFR0_EL1
0b000	0b0000	0b0001	0b100	RO	ID_MMFR0_EL1	ID_MMFR0_EL1
0b000	0b0000	0b0001	0b101	RO	ID_MMFR1_EL1	ID_MMFR1_EL1
0b000	0b0000	0b0001	0b110	RO	ID_MMFR2_EL1	ID_MMFR2_EL1
0b000	0b0000	0b0001	0b111	RO	ID_MMFR3_EL1	ID_MMFR3_EL1
0b000	0b0000	0b0010	0b000	RO	ID_ISAR0_EL1	ID_ISAR0_EL1
0b000	0b0000	0b0010	0b001	RO	ID_ISAR1_EL1	ID_ISAR1_EL1
0b000	0b0000	0b0010	0b010	RO	ID_ISAR2_EL1	ID_ISAR2_EL1
0b000	0b0000	0b0010	0b011	RO	ID_ISAR3_EL1	ID_ISAR3_EL1
0b000	0b0000	0b0010	0b100	RO	ID_ISAR4_EL1	ID_ISAR4_EL1
0b000	0b0000	0b0010	0b101	RO	ID_ISAR5_EL1	ID_ISAR5_EL1
0b000	0b0000	0b0010	0b110	RO	ID_MMFR4_EL1	ID_MMFR4_EL1
0b000	0b0000	0b0010	0b111	RO	ID_ISAR6_EL1	ID_ISAR6_EL1
0b000	0b0000	0b0011	0b000	RO	MVFR0_EL1	MVFR0_EL1
0b000	0b0000	0b0011	0b001	RO	MVFR1_EL1	MVFR1_EL1

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	Access	Mnemonic	Accesses
0b000	0b0000	0b0011	0b010	RO	MVFR2_EL1	MVFR2_EL1
0b000	0b0000	0b0011	0b100	RO	ID_PFR2_EL1	ID_PFR2_EL1
0b000	0b0000	0b0011	0b101	RO	ID_DFR1_EL1	ID_DFR1_EL1
0b000	0b0000	0b0011	0b110	RO	ID_MMFR5_EL1	ID_MMFR5_EL1
0b000	0b0000	0b0100	0b000	RO	ID_AA64PFR0_EL1	ID_AA64PFR0_EL1
0b000	0b0000	0b0100	0b001	RO	ID_AA64PFR1_EL1	ID_AA64PFR1_EL1
0b000	0b0000	0b0100	0b010	RO	ID_AA64PFR2_EL1	ID_AA64PFR2_EL1
0b000	0b0000	0b0101	0b000	RO	ID_AA64DFR0_EL1	ID_AA64DFR0_EL1
0b000	0b0000	0b0101	0b001	RO	ID_AA64DFR1_EL1	ID_AA64DFR1_EL1
0b000	0b0000	0b0101	0b010	RO	ID_AA64DFR2_EL1	ID_AA64DFR2_EL1
0b000	0b0000	0b0101	0b100	RO	ID_AA64AFR0_EL1	ID_AA64AFR0_EL1
0b000	0b0000	0b0101	0b101	RO	ID_AA64AFR1_EL1	ID_AA64AFR1_EL1
0b000	0b0000	0b0110	0b000	RO	ID_AA64ISAR0_EL1	ID_AA64ISAR0_EL1
0b000	0b0000	0b0110	0b001	RO	ID_AA64ISAR1_EL1	ID_AA64ISAR1_EL1
0b000	0b0000	0b0110	0b010	RO	ID_AA64ISAR2_EL1	ID_AA64ISAR2_EL1
0b000	0b0000	0b0111	0b000	RO	ID_AA64MMFR0_EL1	ID_AA64MMFR0_EL1
0b000	0b0000	0b0111	0b001	RO	ID_AA64MMFR1_EL1	ID_AA64MMFR1_EL1
0b000	0b0000	0b0111	0b010	RO	ID_AA64MMFR2_EL1	ID_AA64MMFR2_EL1
0b000	0b0000	0b0111	0b011	RO	ID_AA64MMFR3_EL1	ID_AA64MMFR3_EL1
0b000	0b0000	0b0111	0b100	RO	ID_AA64MMFR4_EL1	ID_AA64MMFR4_EL1
0b000	0b0001	0b0000	0b000	RW	SCTLR_EL1	SCTLR_EL1
0b000	0b0001	0b0000	0b001	RW	ACTLR_EL1	ACTLR_EL1 ACTLR_EL2
0b000	0b0001	0b0000	0b010	RW	CPACR_EL1	CPACR_EL1
0b000	0b0001	0b0010	0b001	RW	TRFCR_EL1	TRFCR_EL1
0b000	0b0010	0b0000	0b000	RW	TTBR0_EL1	TTBR0_EL1
0b000	0b0010	0b0000	0b001	RW	TTBR1_EL1	TTBR1_EL1
0b000	0b0010	0b0000	0b010	RW	TCR_EL1	TCR_EL1
0b000	0b0010	0b0001	0b000	RW	APIAKeyLo_EL1	APIAKeyLo_EL1
0b000	0b0010	0b0001	0b001	RW	APIAKeyHi_EL1	APIAKeyHi_EL1
0b000	0b0010	0b0001	0b010	RW	APIBKeyLo_EL1	APIBKeyLo_EL1
0b000	0b0010	0b0001	0b011	RW	APIBKeyHi_EL1	APIBKeyHi_EL1
0b000	0b0010	0b0010	0b000	RW	APDAKeyLo_EL1	APDAKeyLo_EL1
0b000	0b0010	0b0010	0b001	RW	APDAKeyHi_EL1	APDAKeyHi_EL1
0b000	0b0010	0b0010	0b010	RW	APDBKeyLo_EL1	APDBKeyLo_EL1
0b000	0b0010	0b0010	0b011	RW	APDBKeyHi_EL1	APDBKeyHi_EL1
0b000	0b0010	0b0011	0b000	RW	APGAKeyLo_EL1	APGAKeyLo_EL1
0b000	0b0010	0b0011	0b001	RW	APGAKeyHi_EL1	APGAKeyHi_EL1
0b000	0b0100	0b0000	0b000	RW	SPSR_EL1	SPSR_EL1
0b000	0b0100	0b0000	0b001	RW	ELR_EL1	ELR_EL1

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	Access	Mnemonic	Accesses
0b000	0b0100	0b0001	0b000	RW	SP_EL0	SP_EL0
0b000	0b0100	0b0010	0b000	RW	SPSel	SPSel
0b000	0b0100	0b0010	0b010	RO	CurrentEL	CurrentEL
0b000	0b0100	0b0010	0b011	RW	PAN	PAN
0b000	0b0100	0b0010	0b100	RW	UAO	UAO
0b000	0b0100	0b0110	0b000	RW	ICC_PMR_EL1	ICC_PMR_EL1 ICV_PMR_EL1
0b000	0b0101	0b0001	0b000	RW	AFSR0_EL1	AFSR0_EL1
0b000	0b0101	0b0001	0b001	RW	AFSR1_EL1	AFSR1_EL1
0b000	0b0101	0b0010	0b000	RW	ESR_EL1	ESR_EL1 ESR_EL2
0b000	0b0101	0b0011	0b000	RO	ERRIDR_EL1	ERRIDR_EL1
0b000	0b0101	0b0011	0b001	RW	ERRSELR_EL1	ERRSELR_EL1
0b000	0b0101	0b0100	0b000	RO	ERXFR_EL1	ERXFR_EL1
0b000	0b0101	0b0100	0b001	RW	ERXCTLR_EL1	ERXCTLR_EL1
0b000	0b0101	0b0100	0b010	RW	ERXSTATUS_EL1	ERXSTATUS_EL1
0b000	0b0101	0b0100	0b011	RW	ERXADDR_EL1	ERXADDR_EL1
0b000	0b0101	0b0100	0b100	RO	ERXPFGF_EL1	ERXPFGF_EL1
0b000	0b0101	0b0100	0b101	RW	ERXPFGCTL_EL1	ERXPFGCTL_EL1
0b000	0b0101	0b0100	0b110	RW	ERXPFGCDN_EL1	ERXPFGCDN_EL1
0b000	0b0101	0b0101	0b000	RW	ERXMISC0_EL1	ERXMISC0_EL1
0b000	0b0101	0b0101	0b001	RW	ERXMISC1_EL1	ERXMISC1_EL1
0b000	0b0101	0b0101	0b010	RW	ERXMISC2_EL1	ERXMISC2_EL1
0b000	0b0101	0b0101	0b011	RW	ERXMISC3_EL1	ERXMISC3_EL1
0b000	0b0110	0b0000	0b000	RW	FAR_EL1	FAR_EL1
0b000	0b0110	0b0001	0b001	RW	PRENR_EL1	PRENR_EL1
0b000	0b0110	0b0010	0b001	RW	PRSELR_EL1	PRSELR_EL1
0b000	0b0110	0b1000	0b000	RW	PRBAR_EL1	PRBAR_EL1
0b000	0b0110	0b1000	0b001	RW	PRLAR_EL1	PRLAR_EL1
0b000	0b0110	0b1:m[3:1]	m[0]:0b00	RW	PRBAR<m>_EL1	PRBAR<n>_EL1
0b000	0b0110	0b1:m[3:1]	m[0]:0b01	RW	PRLAR<m>_EL1	PRLAR<n>_EL1
0b000	0b0111	0b0100	0b000	RW	PAR_EL1	PAR_EL1
0b000	0b1001	0b1110	0b001	RW	PMINTENSET_EL1	PMINTENSET_EL1
0b000	0b1001	0b1110	0b010	RW	PMINTENCLR_EL1	PMINTENCLR_EL1
0b000	0b1001	0b1110	0b110	RO	PMMIR_EL1	PMMIR_EL1
0b000	0b1010	0b0010	0b000	RW	MAIR_EL1	MAIR_EL1 MAIR_EL2
0b000	0b1010	0b0011	0b000	RW	AMAIR_EL1	AMAIR_EL1
0b000	0b1100	0b0000	0b000	RW	VBAR_EL1	VBAR_EL1
0b000	0b1100	0b0001	0b000	RO	ISR_EL1	ISR_EL1
0b000	0b1100	0b0001	0b001	RW	DISR_EL1	DISR_EL1 VDISR_EL2

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	Access	Mnemonic	Accesses
0b000	0b1100	0b1000	0b000	RO	ICC_IAR0_EL1	ICC_IAR0_EL1 ICV_IAR0_EL1
0b000	0b1100	0b1000	0b001	WO	ICC_EOIR0_EL1	ICC_EOIR0_EL1 ICV_EOIR0_EL1
0b000	0b1100	0b1000	0b010	RO	ICC_HPPIR0_EL1	ICC_HPPIR0_EL1 ICV_HPPIR0_EL1
0b000	0b1100	0b1000	0b011	RW	ICC_BPR0_EL1	ICC_BPR0_EL1 ICV_BPR0_EL1
0b000	0b1100	0b1000	0b1:m[1:0]	RW	ICC_AP0R<m>_EL1	ICC_AP0R<n>_EL1 ICV_AP0R<n>_EL1
0b000	0b1100	0b1001	0b0:m[1:0]	RW	ICC_AP1R<m>_EL1	ICC_AP1R<n>_EL1 ICV_AP1R<n>_EL1
0b000	0b1100	0b1011	0b001	WO	ICC_DIR_EL1	ICC_DIR_EL1 ICV_DIR_EL1
0b000	0b1100	0b1011	0b011	RO	ICC_RPR_EL1	ICC_RPR_EL1 ICV_RPR_EL1
0b000	0b1100	0b1011	0b101	WO	ICC_SGI1R_EL1	ICC_SGI1R_EL1
0b000	0b1100	0b1011	0b110	WO	ICC_ASGI1R_EL1	ICC_ASGI1R_EL1
0b000	0b1100	0b1011	0b111	WO	ICC_SGI0R_EL1	ICC_SGI0R_EL1
0b000	0b1100	0b1100	0b000	RO	ICC_IAR1_EL1	ICC_IAR1_EL1 ICV_IAR1_EL1
0b000	0b1100	0b1100	0b001	WO	ICC_EOIR1_EL1	ICC_EOIR1_EL1 ICV_EOIR1_EL1
0b000	0b1100	0b1100	0b010	RO	ICC_HPPIR1_EL1	ICC_HPPIR1_EL1 ICV_HPPIR1_EL1
0b000	0b1100	0b1100	0b011	RW	ICC_BPR1_EL1	ICC_BPR1_EL1 ICV_BPR1_EL1
0b000	0b1100	0b1100	0b100	RW	ICC_CTLR_EL1	ICC_CTLR_EL1 ICV_CTLR_EL1
0b000	0b1100	0b1100	0b101	RW	ICC_SRE_EL1	ICC_SRE_EL1
0b000	0b1100	0b1100	0b110	RW	ICC_IGRPEN0_EL1	ICC_IGRPEN0_EL1 ICV_IGRPEN0_EL1
0b000	0b1100	0b1100	0b111	RW	ICC_IGRPEN1_EL1	ICC_IGRPEN1_EL1 ICV_IGRPEN1_EL1
0b000	0b1101	0b0000	0b001	RW	CONTEXTIDR_EL1	CONTEXTIDR_EL1
0b000	0b1101	0b0000	0b100	RW	TPIDR_EL1	TPIDR_EL1
0b000	0b1101	0b0000	0b111	RW	SCXTNUM_EL1	SCXTNUM_EL1 SCXTNUM_EL2
0b000	0b1110	0b0001	0b000	RW	CNTKCTL_EL1	CNTKCTL_EL1
0b001	0b0000	0b0000	0b000	RO	CCSIDR_EL1	CCSIDR_EL1
0b001	0b0000	0b0000	0b001	RO	CLIDR_EL1	CLIDR_EL1
0b001	0b0000	0b0000	0b010	RO	CCSIDR2_EL1	CCSIDR2_EL1
0b001	0b0000	0b0000	0b111	RO	AIDR_EL1	AIDR_EL1
0b010	0b0000	0b0000	0b000	RW	CSSELR_EL1	CSSELR_EL1
0b011	0b0000	0b0000	0b001	RO	CTR_EL0	CTR_EL0
0b011	0b0000	0b0000	0b111	RO	DCZID_EL0	DCZID_EL0
0b011	0b0100	0b0010	0b000	RW	NZCV	NZCV

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	Access	Mnemonic	Accesses
0b011	0b0100	0b0010	0b001	RW	DAIF	DAIF
0b011	0b0100	0b0010	0b101	RW	DIT	DIT
0b011	0b0100	0b0010	0b110	RW	SSBS	SSBS
0b011	0b0100	0b0100	0b000	RW	FPCR	FPCR
0b011	0b0100	0b0100	0b001	RW	FPSR	FPSR
0b011	0b0100	0b0101	0b000	RW	DSPSR_EL0	DSPSR_EL0
0b011	0b0100	0b0101	0b001	RW	DLR_EL0	DLR_EL0
0b011	0b1001	0b1100	0b000	RW	PMCR_EL0	PMCR_EL0
0b011	0b1001	0b1100	0b001	RW	PMCNTENSET_EL0	PMCNTENSET_EL0
0b011	0b1001	0b1100	0b010	RW	PMCNTENCLR_EL0	PMCNTENCLR_EL0
0b011	0b1001	0b1100	0b011	RW	PMOVSCLR_EL0	PMOVSCLR_EL0
0b011	0b1001	0b1100	0b100	WO	PMSWINC_EL0	PMSWINC_EL0
0b011	0b1001	0b1100	0b101	RW	PMSELR_EL0	PMSELR_EL0
0b011	0b1001	0b1100	0b110	RO	PMCEID0_EL0	PMCEID0_EL0
0b011	0b1001	0b1100	0b111	RO	PMCEID1_EL0	PMCEID1_EL0
0b011	0b1001	0b1101	0b000	RW	PMCCNTR_EL0	PMCCNTR_EL0
0b011	0b1001	0b1101	0b001	RW	PMXEVTYPER_EL0	PMXEVTYPER_EL0
0b011	0b1001	0b1101	0b010	RW	PMXEVCNTR_EL0	PMXEVCNTR_EL0
0b011	0b1001	0b1110	0b000	RW	PMUSERENR_EL0	PMUSERENR_EL0
0b011	0b1001	0b1110	0b011	RW	PMOVSSET_EL0	PMOVSSET_EL0
0b011	0b1101	0b0000	0b010	RW	TPIDR_EL0	TPIDR_EL0
0b011	0b1101	0b0000	0b011	RW	TPIDRRO_EL0	TPIDRRO_EL0
0b011	0b1101	0b0000	0b111	RW	SCXTNUM_EL0	SCXTNUM_EL0
0b011	0b1110	0b0000	0b000	RW	CNTFRQ_EL0	CNTFRQ_EL0
0b011	0b1110	0b0000	0b001	RO	CNTPCT_EL0	CNTPCT_EL0
0b011	0b1110	0b0000	0b010	RO	CNTVCT_EL0	CNTVCT_EL0
0b011	0b1110	0b0010	0b000	RW	CNTP_TVAL_EL0	CNTP_TVAL_EL0
0b011	0b1110	0b0010	0b001	RW	CNTP_CTL_EL0	CNTP_CTL_EL0
0b011	0b1110	0b0010	0b010	RW	CNTP_CVAL_EL0	CNTP_CVAL_EL0
0b011	0b1110	0b0011	0b000	RW	CNTV_TVAL_EL0	CNTV_TVAL_EL0
0b011	0b1110	0b0011	0b001	RW	CNTV_CTL_EL0	CNTV_CTL_EL0
0b011	0b1110	0b0011	0b010	RW	CNTV_CVAL_EL0	CNTV_CVAL_EL0
0b011	0b1110	0b10:m[4:3]	m[2:0]	RW	PMEVCNTR<m>_EL0	PMEVCNTR<n>_EL0
0b011	0b1110	0b1111	0b111	RW	PMCCFILTR_EL0	PMCCFILTR_EL0
0b011	0b1110	0b11:m[4:3]	m[2:0]	RW	PMEVTYPER<m>_EL0	PMEVTYPER<n>_EL0
0b100	0b0000	0b0000	0b000	RW	VPIDR_EL2	VPIDR_EL2
0b100	0b0000	0b0000	0b100	RO	MPUIR_EL2	MPUIR_EL2
0b100	0b0000	0b0000	0b101	RW	VMPIDR_EL2	VMPIDR_EL2
0b100	0b0001	0b0000	0b000	RW	SCTLR_EL2	SCTLR_EL2
0b100	0b0001	0b0000	0b001	RW	ACTLR_EL2	ACTLR_EL2

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	Access	Mnemonic	Accesses
0b100	0b0001	0b0001	0b000	RW	HCR_EL2	HCR_EL2
0b100	0b0001	0b0001	0b001	RW	MDCR_EL2	MDCR_EL2
0b100	0b0001	0b0001	0b010	RW	CPTR_EL2	CPTR_EL2
0b100	0b0001	0b0001	0b011	RW	HSTR_EL2	HSTR_EL2
0b100	0b0001	0b0001	0b111	RW	HACR_EL2	HACR_EL2
0b100	0b0001	0b0010	0b001	RW	TRFCR_EL2	TRFCR_EL2
0b100	0b0010	0b0000	0b000	RW	VSCTLR_EL2	VSCTLR_EL2
0b100	0b0010	0b0000	0b010	RW	TCR_EL2	TCR_EL2
0b100	0b0010	0b0001	0b010	RW	VTCR_EL2	VTCR_EL2
0b100	0b0010	0b0110	0b010	RW	VSTCR_EL2	VSTCR_EL2
0b100	0b0100	0b0000	0b000	RW	SPSR_EL2	SPSR_EL2
0b100	0b0100	0b0000	0b001	RW	ELR_EL2	ELR_EL2
0b100	0b0100	0b0001	0b000	RW	SP_EL1	SP_EL1
0b100	0b0100	0b0011	0b000	RW	SPSR_irq	SPSR_irq
0b100	0b0100	0b0011	0b001	RW	SPSR_abt	SPSR_abt
0b100	0b0100	0b0011	0b010	RW	SPSR_und	SPSR_und
0b100	0b0100	0b0011	0b011	RW	SPSR_fiq	SPSR_fiq
0b100	0b0101	0b0001	0b000	RW	AFSR0_EL2	AFSR0_EL2
0b100	0b0101	0b0001	0b001	RW	AFSR1_EL2	AFSR1_EL2
0b100	0b0101	0b0010	0b000	RW	ESR_EL2	ESR_EL1 ESR_EL2
0b100	0b0101	0b0010	0b011	RW	VSESR_EL2	VSESR_EL2
0b100	0b0110	0b0000	0b000	RW	FAR_EL2	FAR_EL2
0b100	0b0110	0b0000	0b100	RW	HPFAR_EL2	HPFAR_EL2
0b100	0b0110	0b0001	0b001	RW	PRENR_EL2	PRENR_EL2
0b100	0b0110	0b0010	0b001	RW	PRSELR_EL2	PRSELR_EL2
0b100	0b0110	0b1000	0b000	RW	PRBAR_EL2	PRBAR_EL2
0b100	0b0110	0b1000	0b001	RW	PRLAR_EL2	PRLAR_EL2
0b100	0b0110	0b1:m[3:1]	m[0]:0b00	RW	PRBAR<m>_EL2	PRBAR<n>_EL2
0b100	0b0110	0b1:m[3:1]	m[0]:0b01	RW	PRLAR<m>_EL2	PRLAR<n>_EL2
0b100	0b1010	0b0010	0b000	RW	MAIR_EL2	MAIR_EL2
0b100	0b1010	0b0011	0b000	RW	AMAIR_EL2	AMAIR_EL2
0b100	0b1100	0b0000	0b000	RW	VBAR_EL2	VBAR_EL2
0b100	0b1100	0b0000	0b001	RO	RVBAR_EL2	RVBAR_EL2
0b100	0b1100	0b0000	0b010	RW	RMR_EL2	RMR_EL2
0b100	0b1100	0b0001	0b001	RW	VDISR_EL2	VDISR_EL2
0b100	0b1100	0b1000	0b0:m[1:0]	RW	ICH_AP0R<m>_EL2	ICH_AP0R<n>_EL2
0b100	0b1100	0b1001	0b0:m[1:0]	RW	ICH_AP1R<m>_EL2	ICH_AP1R<n>_EL2
0b100	0b1100	0b1001	0b101	RW	ICC_SRE_EL2	ICC_SRE_EL2
0b100	0b1100	0b1011	0b000	RW	ICH_HCR_EL2	ICH_HCR_EL2

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op1	CRn	CRm	op2	Access	Mnemonic	Accesses
0b100	0b1100	0b1011	0b001	RO	ICH_VTR_EL2	ICH_VTR_EL2
0b100	0b1100	0b1011	0b010	RO	ICH_MISR_EL2	ICH_MISR_EL2
0b100	0b1100	0b1011	0b011	RO	ICH_EISR_EL2	ICH_EISR_EL2
0b100	0b1100	0b1011	0b101	RO	ICH_ELRSR_EL2	ICH_ELRSR_EL2
0b100	0b1100	0b1011	0b111	RW	ICH_VMCR_EL2	ICH_VMCR_EL2
0b100	0b1100	0b110:m[3]	m[2:0]	RW	ICH_LR<m>_EL2	ICH_LR<n>_EL2
0b100	0b1101	0b0000	0b001	RW	CONTEXTIDR_EL2	CONTEXTIDR_EL2
0b100	0b1101	0b0000	0b010	RW	TPIDR_EL2	TPIDR_EL2
0b100	0b1101	0b0000	0b111	RW	SCXTNUM_EL2	SCXTNUM_EL2
0b100	0b1110	0b0000	0b011	RW	CNTVOFF_EL2	CNTVOFF_EL2
0b100	0b1110	0b0001	0b000	RW	CNTHCTL_EL2	CNTHCTL_EL2
0b100	0b1110	0b0101	0b000	RW	CNTHPS_TVAL_EL2	CNTHPS_TVAL_EL2
0b100	0b1110	0b0101	0b001	RW	CNTHPS_CTL_EL2	CNTHPS_CTL_EL2
0b100	0b1110	0b0101	0b010	RW	CNTHPS_CVAL_EL2	CNTHPS_CVAL_EL2
0b110	0b0100	0b0001	0b000	RW	SP_EL2	SP_EL2

D11.3.1.1 About the GIC System registers

From version 3.0 of the GIC architecture, the specification defines three groups of System registers, identified by the prefix of the register name:

ICC_ GIC physical CPU interface System registers.

ICH_ GIC virtual interface control System registers.

ICV_ GIC Virtual CPU interface System registers.

Note

These registers are in addition to the GIC memory-mapped register groups GICC_, GICD_, GICH_, GICR_, GICV_, and GITS_.

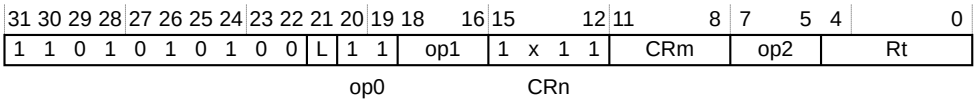
When implemented, the GIC System registers form part of an Arm processor implementation, and therefore these registers are included in the register summaries. However, the registers are defined only in the GIC Architecture Specification.

As Table D11-2 shows, the ICV_* registers have the same {op0, op1, CRn, CRm, op2} encodings as the corresponding ICC_* registers. For these encodings, GIC register configuration fields determine which register is accessed.

For more information, see the *Arm® Generic Interrupt Controller Architecture Specification, GIC architecture version 3.0 and version 4.0* (ARM IHI 0069).

D11.3.2 Reserved encodings for IMPLEMENTATION DEFINED registers

The System register encoding space with op0 == 0b11 reserves the following encodings for IMPLEMENTATION DEFINED registers:



Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The value of `L` defines the access type and the use of `Rt` as follows:

0 Write the value in `Rt` to the IMPLEMENTATION DEFINED register.

1 Read the value of the IMPLEMENTATION DEFINED register to `Rt`.

For more information about these encodings, see S3_<op1>_<Cn>_<Cm>_<op2>, IMPLEMENTATION DEFINED Registers.

As that section describes, any IMPLEMENTATION DEFINED registers are accessed in a similar way to architecturally-defined System registers, using `MRS`, and `MSR`instructions, see:

- MRS.
- MSR (immediate).
- MSR (register).

The Arm architecture guarantees not to define any register name prefixed with *IMP_* as part of the standard Arm architecture.

Note

Arm strongly recommends that any register names created in the IMPLEMENTATION DEFINED register spaces be prefixed with *IMP_* and postfixed with *_ELx*, where appropriate.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter D12
AArch64 System Register Descriptions

This chapter defines the AArch64 System registers. It contains the following sections:

- About the AArch64 System registers.
- General system control registers.
- Debug registers.
- Trace registers.
- Performance Monitors registers.
- PMSA registers
- RAS registers.
- Generic Timer registers.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.1 About the AArch64 System registers

The following sections describe common features of the AArch64 registers:

- Fixed values and reserved values in the System register descriptions.
- General behavior of accesses to the AArch64 System registers.
- Principles of the ID scheme for fields in ID registers.

D12.1.1 Fixed values and reserved values in the System register descriptions

See Fixed values in AArch64 instruction and System register descriptions. This section defines how the glossary terms RAZ, RES0, RAO, and RES1 can be represented in the System register descriptions.

See Reserved values in System and memory-mapped registers and translation table entries. This section describes the behavior of reserved values in the System register descriptions.

D12.1.2 General behavior of accesses to the AArch64 System registers

The following subsections give general information about the behavior of accesses to the System registers:

- Reset behavior of AArch64 System registers.
- Synchronization requirements for AArch64 System registers.

D12.1.2.1 Reset behavior of AArch64 System registers

Reset values apply only to RW registers and fields, however:

- Some RO registers or fields, including feature ID registers and some status registers or register fields, always return a known value.
- Some RW and RO registers or register fields return status information about the PE. Unless the register description indicates that the value is UNKNOWN on reset, a read of the register immediately after a reset returns valid information.
- Some RW and RO registers and fields are aliases of other registers or fields. In these cases, the reset behavior of the aliased register or field determines the value returned by a read of the register immediately after a reset.
- WO registers that only have an effect on writes do not have meaningful reset values. However, an access to a WO register might affect underlying state, and that state might have a defined reset value.
- IMPLEMENTATION DEFINED registers have IMPLEMENTATION DEFINED reset behavior.

After a reset, only a limited subset of the PE state is guaranteed to be set to defined values. Also, for debug and trace System registers, reset requirements must take account of different levels of reset. For more information about the reset behavior of System registers when the PE resets into an Exception level that is using AArch64, see:

- Reset behavior.
- The appropriate Trace architecture specification, for the Trace System registers.

For a PE reset into an Exception level that is using AArch64, the architecture defines which AArch64 System registers have a defined reset value, and when that defined reset value applies. The register descriptions include this information, and Reset behavior summarizes these architectural requirements. Otherwise, RW registers that have a meaningful reset value reset to an architecturally UNKNOWN value.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.1.2.1.1 Pseudocode description of resetting System registers

The `AArch64.ResetSystemRegisters()` pseudocode function resets all System registers, and register fields, that have defined reset values, as described in this section and Reset behavior.

Note

For debug and trace System registers, this function resets registers as defined for the appropriate level of reset.

D12.1.2.2 Synchronization requirements for AArch64 System registers

Reads of the System registers can occur out of order with respect to earlier instructions executed on the same PE, provided that both:

- Any data dependencies between the instructions, including read-after-read dependencies, are respected.
- The reads to the register do not occur earlier than the most recent Context Synchronization event to its architectural position in the instruction stream.

Note

In particular, the values read from System registers that hold self-incrementing counts, such as the Performance Monitors counters or the Generic Timer counter or timers, could be accessed from any time after the previous Context Synchronization event. For example, where a memory access is used to communicate a read of such a counter, an `ISB` must be inserted between the read of the memory location that is known to have returned its data, either as a result of a condition on that data or of the read having completed, and the read of the counter, if it is necessary that the counter returns a count value after the memory communication.

Unless otherwise stated in its System register definition, a direct write to a System register requires synchronization before software can rely on the effects of that write to affect instructions appearing in program order after the write. This does not apply to Special-purpose registers; for more information, see Instructions for accessing Special-purpose registers. Direct writes to System registers are not allowed to affect any instructions appearing in program order before the direct write. The only exceptions are:

- All direct writes to the same register, that use the same encoding for that register, are guaranteed to occur in program order relative to each other.
- All direct writes to a register occur in program order with respect to all direct reads to the same register using the same encoding.
- Any System register access that an Arm *Architecture Specification* or equivalent specification defines as not requiring synchronization.
- When FEAT_BRBE is implemented, execution of `BRB INJ` does not require explicit synchronization to use the result of direct writes to the Branch record injection data registers in program order before `BRB INJ`.

Explicit synchronization occurs as a result of a Context Synchronization event, which is one of the following events:

- Execution of an `ISB` instruction.
- Exception entry, if FEAT_ExS is not implemented, or if FEAT_ExS is implemented and defines that exception entries to this Exception level are Context Synchronization events.
- Exception return, if FEAT_ExS is not implemented, or if FEAT_ExS is implemented and defines that exception returns from this Exception level are Context Synchronization events.
- Execution of a `DCPS` instruction in Debug state.
- Execution of a `DRPS` instruction in Debug state.
- Exit from Debug state.

Note

The `ISB` and exception entry events are applicable both in Debug state and in Non-debug state.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The pseudocode for each of these events defines the point at which the explicit synchronization takes effect, as a result of a call to the `SynchronizeContext()` function or `InstructionSynchronizationBarrier()` function. The principle behind the position of the call to `SynchronizeContext()` or `InstructionSynchronizationBarrier()` is that it occurs before indirect reads of System registers that are used in the construction of the target context, but is permitted to occur after indirect reads of System registers that apply to the context in which the instruction is executed or from which the event is taken.

Note

For some instructions, this means that some System registers are indirectly read before the explicit synchronization, and therefore changes to those System registers might need an explicit Context synchronization event before the instruction. For example, changes to some fields of HCR_EL2 at EL2 need an explicit ISB in program order before an ERET instruction.

In addition, any system instructions that cause a write to a System register must be synchronized before the result is guaranteed to be visible to subsequent direct reads of that System register.

Direct and indirect reads and writes to RGSR_EL1, Random Allocation Tag Seed Register, appear to occur in program order relative to other instructions, without the need for any explicit synchronization.

Direct reads to any one of the following registers, using the same encoding, occur in program order relative to each other:

- ISR_EL1.
- The Generic Timer registers, that is, CNTPCT_EL0 and CNTVCT_EL0, and the Counter registers CNTP_TVAL_EL0, CNTV_TVAL_EL0, CNTHP_TVAL_EL2, and CNTPS_TVAL_EL1.
- DBGCLAIMCLR_EL1.
- The PMU Counters, that is, PMCCNTR_EL0, PMEVCNTR<n>_EL0, PMXEVCNTR_EL0, PMOVSCLR_EL0, and PMOVSSET_EL0.
- The System PMU Counters, that is, SPMEVCNTR<n>_EL0, SPMOVSCLR_EL0, and SPMOVSSET_EL0.
- The Debug Communications Channel registers, that is, DBGDTRRX_EL0, DBGDTR_EL0, and MDCCSR_EL0.

All other direct reads of System registers can occur in any order if synchronization has not been performed.

Table D12-1 describes the synchronization requirements between two successive read or write accesses to the same register, where the ordering of the read or write accesses is:

1. Program order, in the event that both the reads or writes are caused by an instruction executed on this PE, other than one caused by a memory access by this PE.
2. The order of arrival of asynchronous reads and writes at the PE relative to the execution of instructions that cause reads or writes.
3. The order of arrival of asynchronous reads and writes at the PE relative to each other.

Table D12-1 Synchronization requirements

First read/write	Second read/write	Synchronization requirement
Direct read	Direct read	None
	Direct write	None
	Indirect read	None
	Indirect write	None, see Notes
Direct write	Direct read	None
	Direct write	None
	Indirect read	Required
	Indirect write	None, see Notes

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

First read/write	Second read/write	Synchronization requirement
Indirect read	Direct read	None
	Direct write	None
	Indirect read	None
	Indirect write	None
Indirect write	Direct read	Required, see Notes
	Direct write	None, see Notes
	Indirect read	Required, see Notes
	Indirect write	None, see Notes

D12.1.2.2.1 Notes

The terms Direct read, Direct write, Indirect read, and Indirect write, as used in Table D12-1, are defined as follows:

Direct read Where software uses an MRS or MRRS system register access instruction to read that register into a general purpose register.

Where a direct read of a register has a side-effect that changes the contents of a register, the effect of a direct read on that register is defined to be an indirect write. In this case, the indirect write is only guaranteed to have occurred, and be visible to subsequent direct or indirect reads or writes, if synchronization is performed after the direct read.

Direct write Where software uses an MSR (register) or MSRR access instruction to write to that register from a general purpose register.

Where a direct write to a register has an effect on the register that means that the value in the register is not always the last value that is written (as is the case with set and clear registers), the effect of a direct write on that register is defined to be an indirect write. In this case, the indirect write is only guaranteed to be visible to subsequent direct or indirect reads or writes if synchronization is performed after the direct write and before the subsequent direct or indirect reads or writes.

Indirect read Where an instruction uses a System register to establish operating conditions for the instruction, for example, the TTBR_ELx address or whether memory accesses are forced to be Non-cacheable. This includes situations where the contents of one System register selects what value is read or written using a different register. Indirect reads also include reads of the System register by external agents such as debuggers. Instruction execution from MDSTEPOP_EL1 is an indirect read of MDSTEPOP_EL1. Where an indirect read of a register has a side-effect that changes the contents of that register, that is defined to be an indirect write.

Indirect write Where a System register is written as the consequence of some other instruction, exception, operation, or by the asynchronous operation of an external agent, including the passage of time as seen in counters, timers, or performance counters, the assertion of interrupts, or writes from an external debugger.

Note

Registers such as the Exception Syndrome registers that are indirectly written as part of exception entry do not require additional synchronization. This behavior is independent of the value of SCTLR_ELx.EIS, and the list of affected registers is described in the definition of that field.

The value of a system register does not change unless one of the following occurs:

- As the result of a direct write to the register, that is not specified as being trapped, UNDEFINED or IGNORED.
- As the result of one of the specified indirect writes to the register.
- A reset, if the register has a specified reset behavior for that reset.
- The register is set to an UNKNOWN value as the result of one of the above.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Where a direct read or write to a register is followed by an indirect write caused by an external agent, autonomous asynchronous event, or as a result of memory mapped write, synchronization is required to guarantee the order of those two accesses.

Where an indirect write caused by a direct write is followed by an indirect write caused by an external agent, autonomous asynchronous event, or as a result of memory mapped write, synchronization is required to guarantee the order of those two indirect accesses.

Where a direct read to one register causes a bit or field in a different register (or the same register using a different encoding) to be updated, the change to the different register (or same register using a different encoding) is defined to be an indirect write. In this case, the indirect write is only guaranteed to be visible to subsequent direct or indirect reads or writes if synchronization is performed after the direct read and before the subsequent direct or indirect reads or writes.

Where a direct write to one register causes a bit or field in a different register (or the same register using a different encoding) to be updated as a side-effect of that direct write (as opposed to simply being a direct write to the different encoding), the change to the different register (or same register using a different encoding) is defined to be an indirect write. In this case, the indirect write is only guaranteed to be visible to subsequent direct or indirect reads or writes if synchronization is performed after the direct write and before the subsequent direct or indirect reads or writes.

Where indirect writes are caused by the actions of external agents such as debuggers, or by memory-mapped reads or writes by the PE, then an indirect write by that agent and mechanism to a register, followed by an indirect read by that agent and mechanism to the same register using the same address, does not require synchronization.

Where an indirect write occurs as a side-effect of an access, this happens atomically with the access, meaning no other accesses are allowed between the register access and its side-effect.

Note

Even though the 64-bit and 128-bit form of a system register share the same values of `{op0, op1, CRn, CRm, op2}`, they are considered to be distinct encodings. Therefore, explicit synchronization is required to guarantee ordering in all of the following cases:

- MSRR followed by an MRS from the same register.
- MSR (register) followed by an MRRS from the same register.

Where indirect writes to GCSPR_ELx are made by the following instructions, explicit synchronization is not required to be visible to direct reads, indirect reads, and indirect writes in program order after the initial indirect write:

- All Branch with Link instructions.
- All procedure return instructions.
- `GCSPOPM` and `GCSPUSHM`.
- `GCSPOPX`, `GCSPOPCX`, and `GCSPUSHX`.
- `GCSSS1` and `GCSSS2`.

Indirect writes caused by external agents, autonomous asynchronous events, or as a result of memory-mapped writes, to the registers shown in Table D12-2, are required to be observable to:

- Direct reads in finite time without explicit synchronization.
- Subsequent indirect reads without explicit synchronization.

Without explicit synchronization to guarantee the order of the accesses, where the same register is accessed by two or more of a System register access instruction, and external agent, and autonomous asynchronous event, or as a result of a memory-mapped access, the behavior must be as if the accesses occurred atomically and in any order. This applies even if the accesses occur simultaneously.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table D12-2 Registers with a guarantee of observability, VMSAv8-64

Registers	Notes
ISR_EL1	Interrupt Status Register
DBGCLAIMCLR_EL1, DBGCLAIMSET_EL1	Debug CLAIM registers
CNTPCT_EL0, CNTVCT_EL0, CNTP_TVAL_EL0, CNTV_TVAL_EL0, CNTHP_TVAL_EL2, CNTPS_TVAL_EL1	Generic Timer registers
PMCCNTR_EL0, PMEVCNTR<n>_EL0, PMXEVCNTR_EL0, PMOVSCLR_EL0, PMOVSSET_EL0	PMU Counters
SPMEVCNTR<n>_EL0, SPMOVSCLR_EL0, SPMOVSSET_EL0	System PMU Counters
DBGDTRTX_EL0, DBGDTRRX_EL0, DBGDTR_EL0, and the DCC flags in MDCCSR_EL0 and EDSCR	Debug Communication Channel registers
EDSCR.PipeAdv	External Debug Status and Control Register

In addition to the requirements shown in Table D12-2:

- Indirect writes to the following registers as a result of memory-mapped writes, including accesses by external agents, are required to be observable to the indirect read made in determining the response to a subsequent memory-mapped access without explicit synchronization:
  - OSLAR_EL1. OSLAR_EL1 is indirectly read to determine whether the subsequent access is permitted.
  - EDLAR, if implemented. EDLAR is indirectly read to determine whether a subsequent write or side-effect of an access is ignored.

  Note

  This requirement is stricter than the general requirement for the observability of indirect writes.

- The requirement that an indirect write to the registers in Table D12-2 is observable to direct reads in finite time does not imply that all observers will observe the indirect write at the same time.

  For example, an increment of the system counter is an autonomous asynchronous event that performs an indirect write to the counter. This asynchronous event might generate a timer interrupt request, resulting in a Context Synchronization event. When a GIC is used, the timer interrupt might arrive at the GIC after the PE has taken an interrupt request from another source, but before software reads the current interrupt ID from the GIC. This means that the GIC might identify the timer interrupt as the current interrupt. Software must not assume that a subsequent direct read of the counter register is guaranteed to observe the updated value of that register.

  Although this example uses the counter-timer registers, it applies equally to other registers that might be linked to interrupt requests, including the PMU and Statistical Profiling status registers.
- When the PE is in Debug state, there are synchronization requirements for the Debug Communication Channel and Instruction Transfer registers. See DCC and ITR access in Debug state.

Note

- The provision of explicit synchronization requirements to System registers is provided to allow the direct access to these registers to be implemented in a small number of cycles, and that updates to multiple registers can be performed quickly with the synchronization penalty being paid only when the updates have occurred.
- Since toolkits might use registers such as the thread-local storage registers within compiled code, it is recommended that access to these registers is implemented to take a small number of cycles.
- While no synchronization is required between a direct write and a direct read, or between a direct read and an indirect write, this does not imply that a direct read causes synchronization of a previous direct write. That is, the sequence direct write → direct read → indirect read, with no intervening context synchronization, does not guarantee that the indirect read observes the result of the direct write.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.1.3 Principles of the ID scheme for fields in ID registers

The Arm architecture specifies a number of *ID registers* that are characterized as comprising a set of 4-bit *ID fields,* Each ID field identifies the presence, and possibly the level of support for, a particular feature in an implementation of the architecture. These fields follow an architectural model that aids their use by software and provides future compatibility. This section describes that model. ID registers to which this scheme applies identifies the set of ID registers.

A small number of ID fields do not follow the scheme described in this section. In these cases, the field description states that it does not follow this scheme.

Note

- The ID fields described here are unlike the ones that enumerate the number of resources, such as the number of breakpoints, watchpoints, or performance monitors.
- ID fields that do not follow this scheme include the ID_AA64DFR0_EL1.PMUVer, ID_DFR0_EL1.PerfMon, ID_DFR0.PerfMon and EDDFR.PMUVer fields, see Alternative ID scheme used for the Performance Monitors Extension version.
- The presence of an ID field for a feature does not imply that the feature is optional.

To provide forward compatibility, software can rely on the features of these fields that are described in this section.

The ID fields, which are either signed or unsigned, use increasing numerical values to indicate increases in functionality. Therefore, for an unsigned ID field, if the value `0x1` indicates the presence of some instructions, then the value `0x2` will indicate the presence of those instructions plus some additional instructions or functionality. This means software can be written in the form:

```
if (value >= number) {
    // do something that relies on the value of the feature
}
```

For ID fields where the value `0x0` defines that a feature is not present, the field holds an unsigned value. This covers the vast majority of such fields.

In a few cases, the architecture has been changed to permit implementations to exclude a feature that has previously been required and for which no ID field has been defined. In these cases, a new ID field is defined and:

- The field holds a signed value.
- The field value `0x0` indicates that the feature is implemented.
- The field value `0xF` indicates that the feature is not implemented.
- Increasingly negative values indicate additional aspects of the architecture that are not implemented as a result of the feature not being implemented.
- Software that depends on the feature can use the test:

```
if value >= 0 {
    // Software features that depend on the presence of the hardware feature
}
```

In some cases, it has been decided retrospectively that the increase in functionality between two consecutive numerical values is too great, and it is desirable to permit an intermediate degree of functionality, and the means to discover this. This is done by the introduction of a *fractional* field that both:

- Is referred to in the definition of the original field.
- Applies only when the original field is at the lower value of the step.

In principle, a fractional field can be used for two different fractional steps, with different meanings associated with each of these steps. For this reason, a fractional field must be interpreted in the context of the field to which it relates and the value of that field. Example D12-1 shows the use of such a field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example D12-1 Example of the use of a fractional field

For a field describing some class of functionality:

- The value `0x1` was defined as indicating that item A is present.
- The value `0x2` was defined as indicating that items B and C are present, in addition to item A.

Subsequently, it might be necessary to introduce a second ID field to indicate that A and B only are present. This new field is a fractional field, and might be defined as having the value `0x1` when A and B only are present. This fractional field is valid only when the original ID field has the value `0x1`.
This approach means that:

- Software that depends on the test if `(value >= 0x2)` can rely on features A, B, and C being present,
- Software that depends on the test `if (value >= 0x1)` can rely on feature A being present.
- If new software needs to check only that features A and B are present, then it can test:

```
if (value >= 0x2 || (value == 0x1 && fractional_value >= 0x1)) {
    // Software features that depend on A and B only
}
```

A fractional field uses the same approach of increasing numerical values indicating increasing functionality, and the fractional approach can also be applied recursively to fractional fields.

Unused ID fields, and fractional fields that are not applicable, are RES0 to allow their future use when features, or fractional implementation options, are added.

D12.1.3.1 ID registers to which this scheme applies

This scheme applies to the following registers:

AArch64 System registers

- The Auxiliary Feature register ID_AFR0_EL1.
- The Processor Feature registers ID_PFR0_EL1, ID_PFR1_EL1, and ID_PFR2_EL1.
- The Debug Feature register ID_DFR0_EL1 and ID_DFR1_EL1.
- The Memory Model Feature registers ID_MMFR0_EL1, ID_MMFR1_EL1, ID_MMFR2_EL1, ID_MMFR3_EL1, ID_MMFR4_EL1, and ID_MMFR5_EL1.
- The Instruction Set Attribute registers ID_ISAR0_EL1, ID_ISAR1_EL1, ID_ISAR2_EL1, ID_ISAR3_EL1, ID_ISAR4_EL1, ID_ISAR5_EL1, and ID_ISAR6_EL1.
- The Media and VFP Feature registers MVFR0_EL1, MVFR1_EL1, and MVFR2_EL1.
- The AArch64 Auxiliary Feature registers ID_AA64AFR0_EL1 and ID_AA64AFR1_EL1.
- The AArch64 Processor Feature registers ID_AA64PFR0_EL1, ID_AA64PFR1_EL1, and ID_AA64PFR2_EL1.
- The AArch64 Debug Feature registers ID_AA64DFR0_EL1, ID_AA64DFR1_EL1, and ID_AA64DFR2_EL1.
- The AArch64 Memory Model Feature registers ID_AA64MMFR0_EL1, ID_AA64MMFR1_EL1, ID_AA64MMFR2_EL1, ID_AA64MMFR3_EL1, and ID_AA64MMFR4_EL1.
- The AArch64 Instruction Set Attribute registers ID_AA64ISAR0_EL1, ID_AA64ISAR1_EL1, ID_AA64ISAR2_EL1, and ID_AA64ISAR3_EL1.
- The AArch64 SVE Feature ID register ID_AA64ZFR0_EL1.

AArch64 System registers

- The Auxiliary Feature register ID_AFR0_EL1.
- The Processor Feature registers ID_PFR0_EL1, ID_PFR1_EL1, and ID_PFR2_EL1.
- The Debug Feature register ID_DFR0_EL1 and ID_DFR1_EL1.
- The Memory Model Feature registers ID_MMFR0_EL1, ID_MMFR1_EL1, ID_MMFR2_EL1, ID_MMFR3_EL1, ID_MMFR4_EL1, and ID_MMFR5_EL1.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The Instruction Set Attribute registers ID_ISAR0_EL1, ID_ISAR1_EL1, ID_ISAR2_EL1, ID_ISAR3_EL1, ID_ISAR4_EL1, ID_ISAR5_EL1, and ID_ISAR6_EL1.
- The Media and VFP Feature registers MVFR0_EL1, MVFR1_EL1, and MVFR2_EL1.
- The AArch64 Auxiliary Feature registers ID_AA64AFR0_EL1 and ID_AA64AFR1_EL1.
- The AArch64 Processor Feature registers ID_AA64PFR0_EL1, ID_AA64PFR1_EL1, and ID_AA64PFR2_EL1.
- The AArch64 Debug Feature registers ID_AA64DFR0_EL1, ID_AA64DFR1_EL1, and ID_AA64DFR2_EL1.
- The AArch64 Memory Model Feature registers ID_AA64MMFR0_EL1, ID_AA64MMFR1_EL1, ID_AA64MMFR2_EL1, ID_AA64MMFR3_EL1, and ID_AA64MMFR4_EL1.
- The AArch64 Instruction Set Attribute registers ID_AA64ISAR0_EL1, ID_AA64ISAR1_EL1, ID_AA64ISAR2_EL1.

Memory-mapped registers

- The External Debug Processor Feature register EDPFR.
- The External Debug Feature register EDDFR.
- The External Debug Auxiliary Processor Feature register EDAA32PFR.

D12.1.3.2 Alternative ID scheme used for the Performance Monitors Extension version

The ID_AA64DFR0_EL1.PMUVer, ID_DFR0_EL1.PerfMon, ID_DFR0.PerfMon and EDDFR.PMUVer fields, that identify the version of the Performance Monitors Extension, do not follow the standard ID scheme. Software must treat these fields as follows:

- The value `0xF` indicates that the Arm-architected Performance Monitors Extension is not implemented.
- If the field value is not `0xF`, the field is treated as an unsigned value, as described for the standard ID scheme.

This means that software that depends on the implementation of a particular version of the Arm Performance Monitors Extension must be written in the form:

```
if (value != 0xF and value >= number) {
    // do something that relies on version 'number' of the feature
}
```

For these fields, Arm deprecates use of the value `0xF` in new implementations.

D12.1.3.3 Alternative ID scheme used for ID_AA64MMFR0_EL1 stage 2 granule sizes

The ID_AA64MMFR0_EL1.{TGran4_2, TGran64_2, TGran16_2} fields that identify the memory translation stage 2 granule size, do not follow the standard ID scheme. Software must treat these fields as follows:

- The value `0x0` indicates that support is identified by another field.
- If the field value is not `0x0`, the value indicates the level of support provided.

This means that software should use a test of the form:

```
if (field !=0 and field > value) {
    // do something that relies on the value of the feature
}
```

D12.1.3.4 Alternative ID scheme used for ID_AA64SMFR0_EL1 and ID_AA64FPFR0_EL1

Apart from the ID_AA64SMFR0_EL1.SMEver field, which is a 4-bit unsigned integer conforming to the standard scheme, software must treat the other fields in these registers as follows:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- A 1-bit field value where the bit is `0b0` indicates that the specified feature or instructions described by this field are not implemented.
- A 1-bit field value where the bit is `0b1` indicates that the specified feature or instructions described by this field are implemented.
- In ID_AA64SMFR0_EL1, a 4-bit field indicates whether a group of related SME instructions is implemented, with permitted values defined in the field description. Bits within such a field which only permit the value `0b0000` might be used to identify new instructions in a future version of SME, without changing the meaning of those bits that permit the value `0b0001`.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2 General system control registers

This section lists the System registers in AArch64 that are not part of one of the other listed groups.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.1 ACTLR_EL1, Auxiliary Control Register (EL1)

The ACTLR_EL1 characteristics are:

Purpose

Provides IMPLEMENTATION DEFINED configuration and control options for execution at EL1 and EL0.

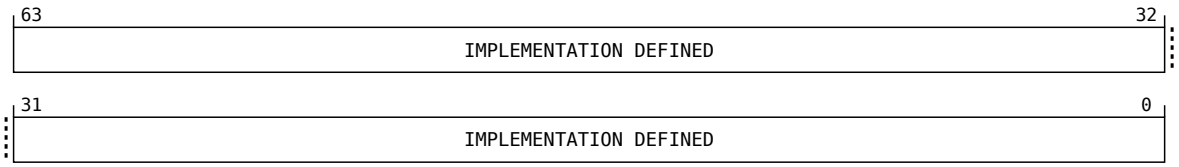
Configuration

There are no configuration notes.

Attributes

ACTLR_EL1 is a 64-bit register.

Field descriptions



IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing ACTLR_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, ACTLR_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TACR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ACTLR_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ACTLR_EL1;
```

`MSR ACTLR_EL1, <Xt>`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b001

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TACR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        ACTLR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    ACTLR_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.2 ACTLR_EL2, Auxiliary Control Register (EL2)

The ACTLR_EL2 characteristics are:

Purpose

Provides IMPLEMENTATION DEFINED configuration and control options for EL2.

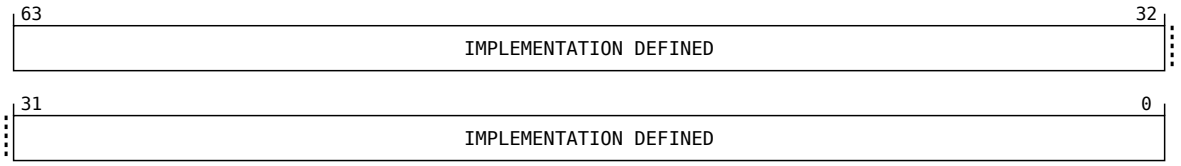
Configuration

There are no configuration notes.

Attributes

ACTLR_EL2 is a 64-bit register.

Field descriptions



IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing ACTLR_EL2

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, ACTLR_EL2`

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ACTLR_EL2;
```

`MSR ACTLR_EL2, <Xt>`

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    ACTLR_EL2 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When an implementation implements ACTLR_ELx accessor behavior MRS <Xt>, ACTLR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TACR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ACTLR_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ACTLR_EL1;
```

When an implementation implements ACTLR_ELx accessor behavior MSR ACTLR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TACR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        ACTLR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    ACTLR_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.3 AFSR0_EL1, Auxiliary Fault Status Register 0 (EL1)

The AFSR0_EL1 characteristics are:

Purpose

Provides additional IMPLEMENTATION DEFINED fault status information for exceptions taken to EL1.

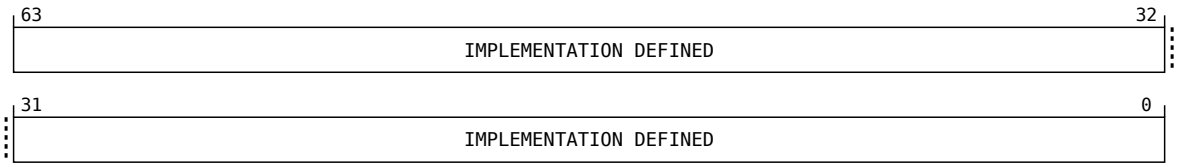
Configuration

There are no configuration notes.

Attributes

AFSR0_EL1 is a 64-bit register.

Field descriptions



IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing AFSR0_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, AFSR0_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = AFSR0_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = AFSR0_EL1;
```

`MSR AFSR0_EL1, <Xt>`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0001	0b000

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AFSR0_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    AFSR0_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.4 AFSR0_EL2, Auxiliary Fault Status Register 0 (EL2)

The AFSR0_EL2 characteristics are:

Purpose

Provides additional IMPLEMENTATION DEFINED fault status information for exceptions taken to EL2.

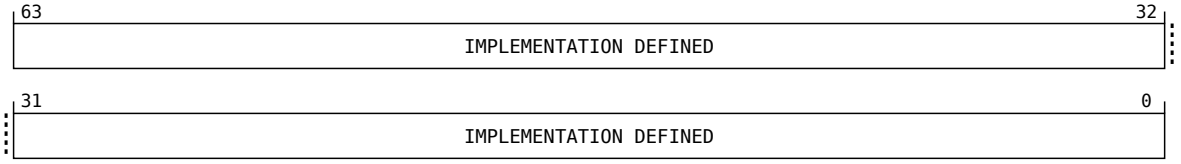
Configuration

There are no configuration notes.

Attributes

AFSR0_EL2 is a 64-bit register.

Field descriptions



IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing AFSR0_EL2

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, AFSR0_EL2`

op0	op1	CRn	CRm	op2
0b11	0b100	0b0101	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = AFSR0_EL2;
```

`MSR AFSR0_EL2, <Xt>`

op0	op1	CRn	CRm	op2
0b11	0b100	0b0101	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    AFSR0_EL2 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.5 AFSR1_EL1, Auxiliary Fault Status Register 1 (EL1)

The AFSR1_EL1 characteristics are:

Purpose

Provides additional IMPLEMENTATION DEFINED fault status information for exceptions taken to EL1.

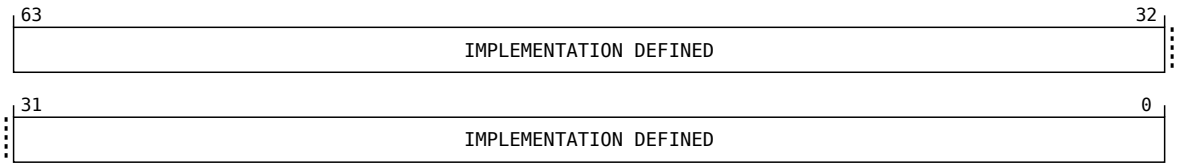
Configuration

There are no configuration notes.

Attributes

AFSR1_EL1 is a 64-bit register.

Field descriptions



IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing AFSR1_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, AFSR1_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = AFSR1_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = AFSR1_EL1;
```

`MSR AFSR1_EL1, <Xt>`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0001	0b001

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AFSR1_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    AFSR1_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.6 AFSR1_EL2, Auxiliary Fault Status Register 1 (EL2)

The AFSR1_EL2 characteristics are:

Purpose

Provides additional IMPLEMENTATION DEFINED fault status information for exceptions taken to EL2.

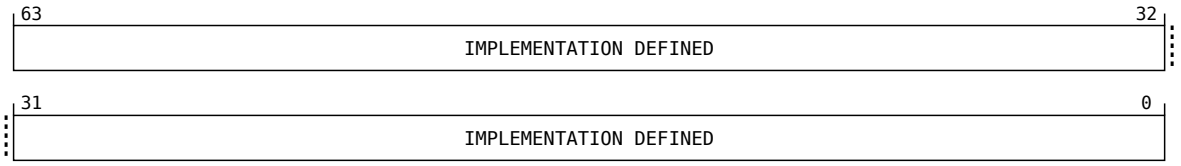
Configuration

There are no configuration notes.

Attributes

AFSR1_EL2 is a 64-bit register.

Field descriptions



IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing AFSR1_EL2

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, AFSR1_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0101	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = AFSR1_EL2;
```

MSR AFSR1_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0101	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    AFSR1_EL2 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.7 AIDR_EL1, Auxiliary ID Register

The AIDR_EL1 characteristics are:

Purpose

Provides IMPLEMENTATION DEFINED identification information.

The value of this register must be interpreted in conjunction with the value of MIDR_EL1.

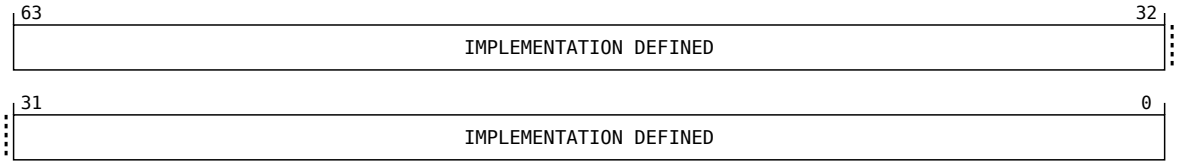
Configuration

There are no configuration notes.

Attributes

AIDR_EL1 is a 64-bit register.

Field descriptions



IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

Accessing AIDR_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, AIDR_EL1`

op0	op1	CRn	CRm	op2
0b11	0b001	0b0000	0b0000	0b111

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TID1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = AIDR_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = AIDR_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.8 AMAIR_EL1, Auxiliary Memory Attribute Indirection Register (EL1)

The AMAIR_EL1 characteristics are:

Purpose

Provides IMPLEMENTATION DEFINED memory attributes for the memory regions specified by MAIR_EL1.

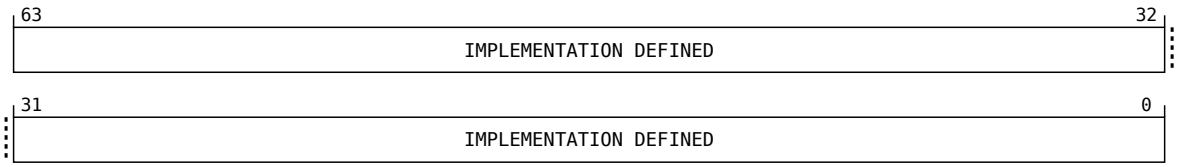
Configuration

There are no configuration notes.

Attributes

AMAIR_EL1 is a 64-bit register.

Field descriptions



AMAIR_EL1 is permitted to be cached in a TLB.

IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing AMAIR_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, AMAIR_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0011	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = AMAIR_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = AMAIR_EL1;
```

`MSR AMAIR_EL1, <Xt>`

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0011	0b000

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AMAIR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    AMAIR_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.9 AMAIR_EL2, Auxiliary Memory Attribute Indirection Register (EL2)

The AMAIR_EL2 characteristics are:

Purpose

Provides IMPLEMENTATION DEFINED memory attributes for the memory regions specified by MAIR_EL2.

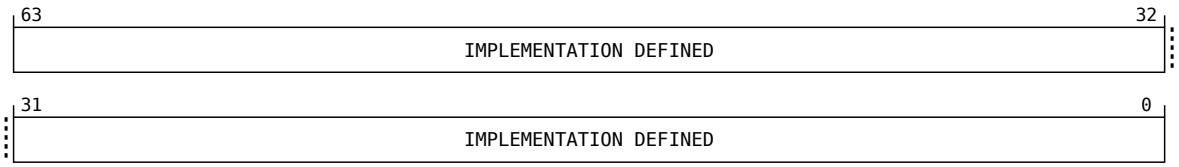
Configuration

There are no configuration notes.

Attributes

AMAIR_EL2 is a 64-bit register.

Field descriptions



AMAIR_EL2 is permitted to be cached in a TLB.

IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing AMAIR_EL2

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, AMAIR_EL2`

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0011	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = AMAIR_EL2;
```

`MSR AMAIR_EL2, <Xt>`

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0011	0b000

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    AMAIR_EL2 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.10 APDAKeyHi_EL1, Pointer Authentication Key A for Data (bits[127:64])

The APDAKeyHi_EL1 characteristics are:

Purpose

Holds bits[127:64] of key A used for authentication of data pointer values.

Note

The term APDAKey_EL1 is used to describe the concatenation of APDAKeyHi_EL1: APDAKeyLo_EL1.

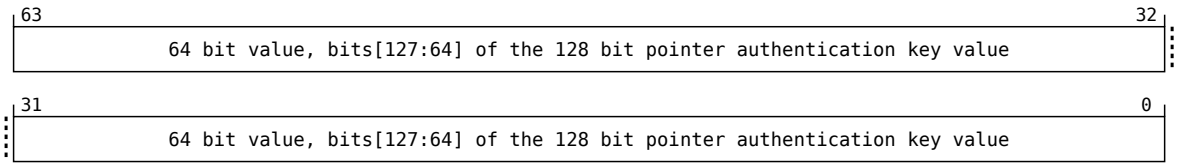
Configuration

This register is present only when FEAT_PAuth is implemented. Otherwise, direct accesses to APDAKeyHi_EL1 are UNDEFINED.

Attributes

APDAKeyHi_EL1 is a 64-bit register.

Field descriptions



APDAKeyHi, bits [63:0]

64 bit value, bits[127:64] of the 128 bit pointer authentication key value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing APDAKeyHi_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, APDAKeyHi_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0010	0b001

```
if !IsFeatureImplemented(FEAT_PAuth) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.APK == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = APDAKeyHi_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = APDAKeyHi_EL1;
```

`MSR APDAKeyHi_EL1, <Xt>`

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0010	0b001

```
if !IsFeatureImplemented(FEAT_PAuth) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.APK == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        APDAKeyHi_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    APDAKeyHi_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.11 APDAKeyLo_EL1, Pointer Authentication Key A for Data (bits[63:0])

The APDAKeyLo_EL1 characteristics are:

Purpose

Holds bits[63:0] of key A used for authentication of data pointer values.

Note

The term APDAKey_EL1 is used to describe the concatenation of APDAKeyHi_EL1: APDAKeyLo_EL1.

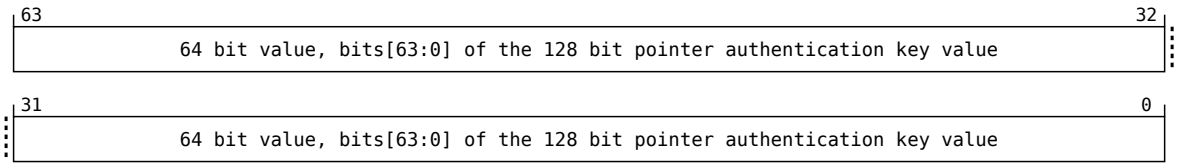
Configuration

This register is present only when FEAT_PAuth is implemented. Otherwise, direct accesses to APDAKeyLo_EL1 are UNDEFINED.

Attributes

APDAKeyLo_EL1 is a 64-bit register.

Field descriptions



APDAKeyLo, bits [63:0]

64 bit value, bits[63:0] of the 128 bit pointer authentication key value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing APDAKeyLo_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, APDAKeyLo_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_PAuth) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.APK == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = APDAKeyLo_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = APDAKeyLo_EL1;
```

`MSR APDAKeyLo_EL1, <Xt>`

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0010	0b000

```
if !IsFeatureImplemented(FEAT_PAuth) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.APK == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        APDAKeyLo_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    APDAKeyLo_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.12 APDBKeyHi_EL1, Pointer Authentication Key B for Data (bits[127:64])

The APDBKeyHi_EL1 characteristics are:

Purpose

Holds bits[127:64] of key B used for authentication of data pointer values.

Note

The term APDBKey_EL1 is used to describe the concatenation of APDBKeyHi_EL1: APDBKeyLo_EL1.

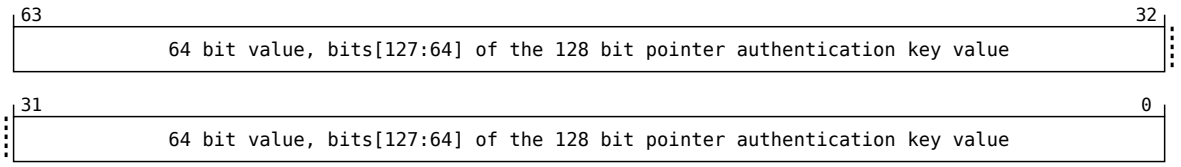
Configuration

This register is present only when FEAT_PAuth is implemented. Otherwise, direct accesses to APDBKeyHi_EL1 are UNDEFINED.

Attributes

APDBKeyHi_EL1 is a 64-bit register.

Field descriptions



APDBKeyHi, bits [63:0]

64 bit value, bits[127:64] of the 128 bit pointer authentication key value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing APDBKeyHi_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, APDBKeyHi_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0010	0b011

```
if !IsFeatureImplemented(FEAT_PAuth) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.APK == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = APDBKeyHi_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = APDBKeyHi_EL1;
```

`MSR APDBKeyHi_EL1, <Xt>`

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0010	0b011

```
if !IsFeatureImplemented(FEAT_PAuth) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.APK == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        APDBKeyHi_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    APDBKeyHi_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.13 APDBKeyLo_EL1, Pointer Authentication Key B for Data (bits[63:0])

The APDBKeyLo_EL1 characteristics are:

Purpose

Holds bits[63:0] of key B used for authentication of data pointer values.

Note

The term APDBKey_EL1 is used to describe the concatenation of APDBKeyHi_EL1: APDBKeyLo_EL1.

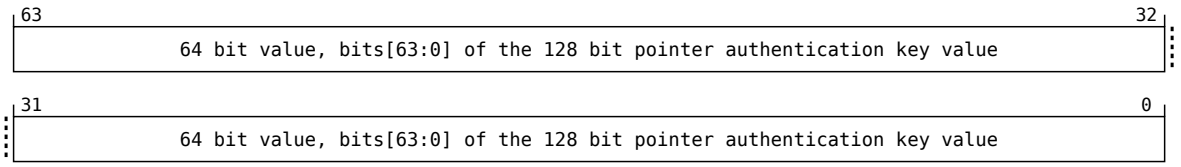
Configuration

This register is present only when FEAT_PAuth is implemented. Otherwise, direct accesses to APDBKeyLo_EL1 are UNDEFINED.

Attributes

APDBKeyLo_EL1 is a 64-bit register.

Field descriptions



APDBKeyLo, bits [63:0]

64 bit value, bits[63:0] of the 128 bit pointer authentication key value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing APDBKeyLo_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, APDBKeyLo_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0010	0b010

```
if !IsFeatureImplemented(FEAT_PAuth) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.APK == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = APDBKeyLo_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = APDBKeyLo_EL1;
```

`MSR APDBKeyLo_EL1, <Xt>`

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0010	0b010

```
if !IsFeatureImplemented(FEAT_PAuth) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.APK == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        APDBKeyLo_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    APDBKeyLo_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.14 APGAKeyHi_EL1, Pointer Authentication Key A for Code (bits[127:64])

The APGAKeyHi_EL1 characteristics are:

Purpose

Holds bits[127:64] of key used for generic pointer authentication code.

Note

The term APGAKey_EL1 is used to describe the concatenation of APGAKeyHi_EL1: APGAKeyLo_EL1.

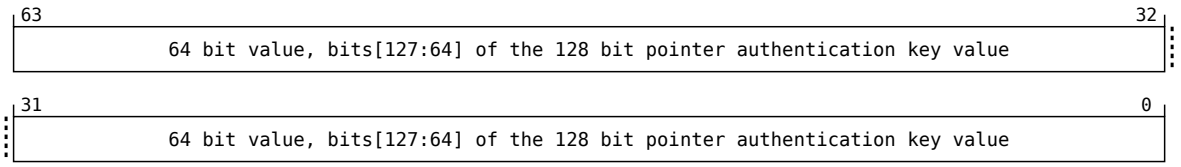
Configuration

This register is present only when FEAT_PAuth is implemented. Otherwise, direct accesses to APGAKeyHi_EL1 are UNDEFINED.

Attributes

APGAKeyHi_EL1 is a 64-bit register.

Field descriptions



APGAKeyHi, bits [63:0]

64 bit value, bits[127:64] of the 128 bit pointer authentication key value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing APGAKeyHi_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, APGAKeyHi_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0011	0b001

```
if !IsFeatureImplemented(FEAT_PAuth) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.APK == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = APGAKeyHi_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = APGAKeyHi_EL1;
```

`MSR APGAKeyHi_EL1, <Xt>`

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0011	0b001

```
if !IsFeatureImplemented(FEAT_PAuth) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.APK == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        APGAKeyHi_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    APGAKeyHi_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.15 APGAKeyLo_EL1, Pointer Authentication Key A for Code (bits[63:0])

The APGAKeyLo_EL1 characteristics are:

Purpose

Holds bits[63:0] of key used for generic pointer authentication code.

Note

The term APGAKey_EL1 is used to describe the concatenation of APGAKeyHi_EL1: APGAKeyLo_EL1.

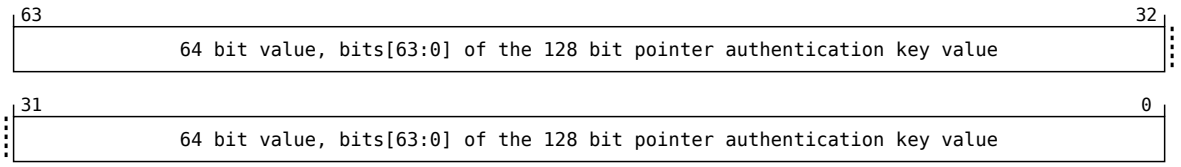
Configuration

This register is present only when FEAT_PAuth is implemented. Otherwise, direct accesses to APGAKeyLo_EL1 are UNDEFINED.

Attributes

APGAKeyLo_EL1 is a 64-bit register.

Field descriptions



APGAKeyLo, bits [63:0]

64 bit value, bits[63:0] of the 128 bit pointer authentication key value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing APGAKeyLo_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, APGAKeyLo_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0011	0b000

```
if !IsFeatureImplemented(FEAT_PAuth) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.APK == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = APGAKeyLo_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = APGAKeyLo_EL1;
```

`MSR APGAKeyLo_EL1, <Xt>`

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0011	0b000

```
if !IsFeatureImplemented(FEAT_PAuth) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.APK == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        APGAKeyLo_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    APGAKeyLo_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.16 APIAKeyHi_EL1, Pointer Authentication Key A for Instruction (bits[127:64])

The APIAKeyHi_EL1 characteristics are:

Purpose

Holds bits[127:64] of key A used for authentication of instruction pointer values.

Note

The term APIAKey_EL1 is used to describe the concatenation of APIAKeyHi_EL1: APIAKeyLo_EL1.

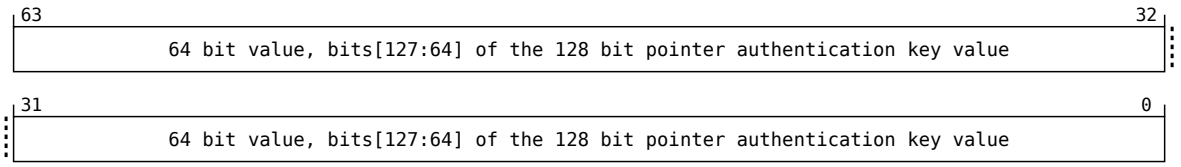
Configuration

This register is present only when FEAT_PAuth is implemented. Otherwise, direct accesses to APIAKeyHi_EL1 are UNDEFINED.

Attributes

APIAKeyHi_EL1 is a 64-bit register.

Field descriptions



APIAKeyHi, bits [63:0]

64 bit value, bits[127:64] of the 128 bit pointer authentication key value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing APIAKeyHi_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, APIAKeyHi_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0001	0b001

```
if !IsFeatureImplemented(FEAT_PAuth) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.APK == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = APIAKeyHi_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = APIAKeyHi_EL1;
```

`MSR APIAKeyHi_EL1, <Xt>`

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0001	0b001

```
if !IsFeatureImplemented(FEAT_PAuth) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.APK == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        APIAKeyHi_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    APIAKeyHi_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.17 APIAKeyLo_EL1, Pointer Authentication Key A for Instruction (bits[63:0])

The APIAKeyLo_EL1 characteristics are:

Purpose

Holds bits[63:0] of key A used for authentication of instruction pointer values.

Note

The term APIAKey_EL1 is used to describe the concatenation of APIAKeyHi_EL1: APIAKeyLo_EL1.

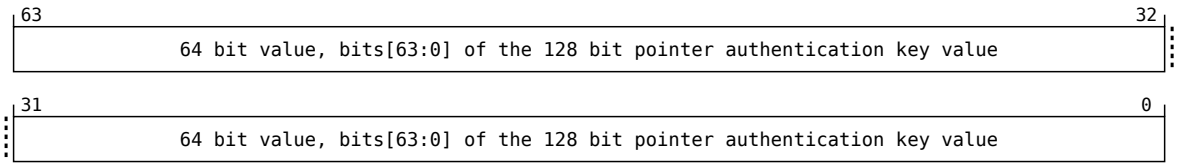
Configuration

This register is present only when FEAT_PAuth is implemented. Otherwise, direct accesses to APIAKeyLo_EL1 are UNDEFINED.

Attributes

APIAKeyLo_EL1 is a 64-bit register.

Field descriptions



APIAKeyLo, bits [63:0]

64 bit value, bits[63:0] of the 128 bit pointer authentication key value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing APIAKeyLo_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, APIAKeyLo_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_PAuth) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.APK == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = APIAKeyLo_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = APIAKeyLo_EL1;
```

`MSR APIAKeyLo_EL1, <Xt>`

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0001	0b000

```
if !IsFeatureImplemented(FEAT_PAuth) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.APK == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        APIAKeyLo_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    APIAKeyLo_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.18 APIBKeyHi_EL1, Pointer Authentication Key B for Instruction (bits[127:64])

The APIBKeyHi_EL1 characteristics are:

Purpose

Holds bits[127:64] of key B used for authentication of instruction pointer values.

Note

The term APIBKey_EL1 is used to describe the concatenation of APIBKeyHi_EL1: APIBKeyLo_EL1.

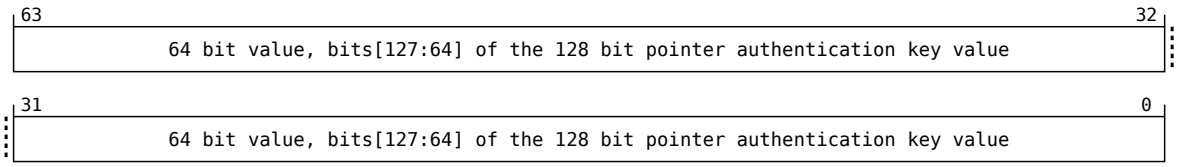
Configuration

This register is present only when FEAT_PAuth is implemented. Otherwise, direct accesses to APIBKeyHi_EL1 are UNDEFINED.

Attributes

APIBKeyHi_EL1 is a 64-bit register.

Field descriptions



APIBKeyHi, bits [63:0]

64 bit value, bits[127:64] of the 128 bit pointer authentication key value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing APIBKeyHi_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, APIBKeyHi_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0001	0b011

```
if !IsFeatureImplemented(FEAT_PAuth) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.APK == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = APIBKeyHi_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = APIBKeyHi_EL1;
```

`MSR APIBKeyHi_EL1, <Xt>`

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0001	0b011

```
if !IsFeatureImplemented(FEAT_PAuth) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.APK == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        APIBKeyHi_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    APIBKeyHi_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.19 APIBKeyLo_EL1, Pointer Authentication Key B for Instruction (bits[63:0])

The APIBKeyLo_EL1 characteristics are:

Purpose

Holds bits[63:0] of key B used for authentication of instruction pointer values.

Note

The term APIBKey_EL1 is used to describe the concatenation of APIBKeyHi_EL1: APIBKeyLo_EL1.

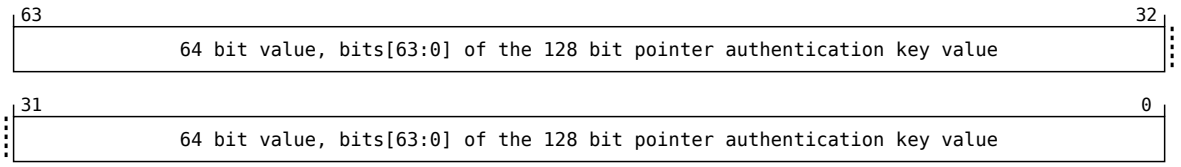
Configuration

This register is present only when FEAT_PAuth is implemented. Otherwise, direct accesses to APIBKeyLo_EL1 are UNDEFINED.

Attributes

APIBKeyLo_EL1 is a 64-bit register.

Field descriptions



APIBKeyLo, bits [63:0]

64 bit value, bits[63:0] of the 128 bit pointer authentication key value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing APIBKeyLo_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, APIBKeyLo_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0001	0b010

```
if !IsFeatureImplemented(FEAT_PAuth) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.APK == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = APIBKeyLo_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = APIBKeyLo_EL1;
```

`MSR APIBKeyLo_EL1, <Xt>`

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0001	0b010

```
if !IsFeatureImplemented(FEAT_PAuth) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.APK == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        APIBKeyLo_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    APIBKeyLo_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.20 CCSIDR2_EL1, Current Cache Size ID Register 2

The CCSIDR2_EL1 characteristics are:

Purpose

Provides the information about the architecture of the currently selected cache from bits[63:32] of CCSIDR_EL1.

Configuration

In an implementation which does not support AArch32 at EL1, it is IMPLEMENTATION DEFINED whether reading this register gives an UNKNOWN value or is UNDEFINED.

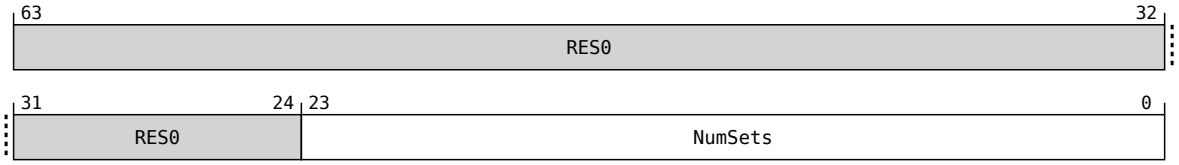
The implementation includes one CCSIDR2_EL1 for each cache that it can access. CSSELR_EL1 selects which Cache Size ID Register is accessible.

This register is present only when FEAT_CCIDX is implemented. Otherwise, direct accesses to CCSIDR2_EL1 are UNDEFINED.

Attributes

CCSIDR2_EL1 is a 64-bit register.

Field descriptions



Bits [63:24]

Reserved, RES0.

NumSets, bits [23:0]

(Number of sets in cache) - 1, therefore a value of 0 indicates 1 set in the cache. The number of sets does not have to be a power of 2.

Accessing CCSIDR2_EL1

If CSSELR_EL1.{TnD, Level, InD} is programmed to a cache level that is not implemented, then on a read of the CCSIDR2_EL1 the behavior is CONSTRAINED UNPREDICTABLE, and can be one of the following:

- The CCSIDR2_EL1 read is treated as NOP.
- The CCSIDR2_EL1 read is UNDEFINED. If FEAT_IDST is implemented, this is permitted to be reported with EC syndrome value 0x18.
- The CCSIDR2_EL1 read returns an UNKNOWN value.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, CCSIDR2_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b001	0b0000	0b0000	0b010

```
if !IsFeatureImplemented(FEAT_CCIDX) then
    UnimplementedIDRegister();
elsif PSTATE.EL == EL0 then
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if IsFeatureImplemented(FEAT_IDST) then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TID2 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = CCSIDR2_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = CCSIDR2_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.21 CCSIDR_EL1, Current Cache Size ID Register

The CCSIDR_EL1 characteristics are:

Purpose

Provides information about the architecture of the currently selected cache.

Configuration

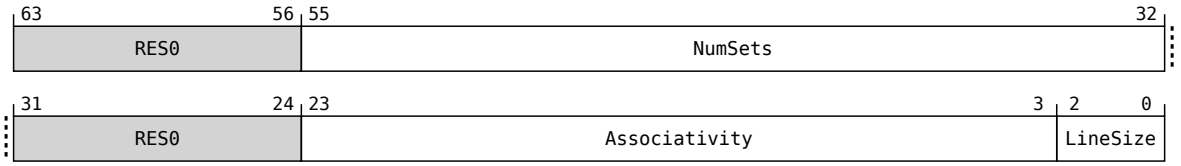
The implementation includes one CCSIDR_EL1 for each cache that it can access. CSSELR_EL1 selects which Cache Size ID Register is accessible.

Attributes

CCSIDR_EL1 is a 64-bit register.

Field descriptions

When FEAT_CCIDX is implemented:



Note

The parameters NumSets, Associativity, and LineSize in these registers define the architecturally visible parameters that are required for the cache maintenance by Set/Way instructions. They are not guaranteed to represent the actual microarchitectural features of a design. You cannot make any inference about the actual sizes of caches based on these parameters.

Bits [63:56]

Reserved, RES0.

NumSets, bits [55:32]

(Number of sets in cache) - 1, therefore a value of 0 indicates 1 set in the cache. The number of sets does not have to be a power of 2.

Bits [31:24]

Reserved, RES0.

Associativity, bits [23:3]

(Associativity of cache) - 1, therefore a value of 0 indicates an associativity of 1. The associativity does not have to be a power of 2.

LineSize, bits [2:0]

(Log_2(Number of bytes in cache line)) - 4. For example:

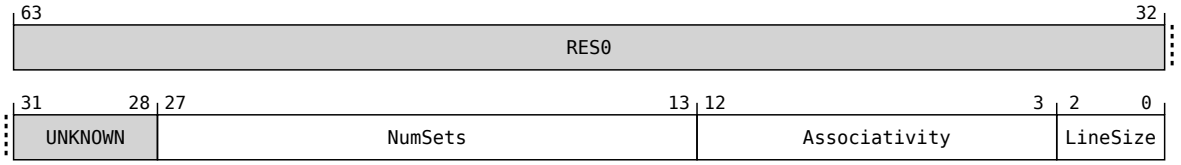
- For a line length of 16 bytes: $Log_2(16) = 4$, LineSize entry = 0. This is the minimum line length.
- For a line length of 32 bytes: $Log_2(32) = 5$, LineSize entry = 1.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

The C++ 17 specification has two defined parameters relating to the granularity of memory that does not interfere. For generic software and tools, Arm will set the hardware_destructive_interference_size parameter to 256 bytes and the hardware_constructive_interference_size parameter to 64 bytes.

Otherwise:



Note

The parameters NumSets, Associativity, and LineSize in these registers define the architecturally visible parameters that are required for the cache maintenance by Set/Way instructions. They are not guaranteed to represent the actual microarchitectural features of a design. You cannot make any inference about the actual sizes of caches based on these parameters.

Bits [63:32]

Reserved, RES0.

Bits [31:28]

Reserved, UNKNOWN.

NumSets, bits [27:13]

(Number of sets in cache) - 1, therefore a value of 0 indicates 1 set in the cache. The number of sets does not have to be a power of 2.

Associativity, bits [12:3]

(Associativity of cache) - 1, therefore a value of 0 indicates an associativity of 1. The associativity does not have to be a power of 2.

LineSize, bits [2:0]

(Log_2(Number of bytes in cache line)) - 4. For example:

- For a line length of 16 bytes: $Log_2(16) = 4$, LineSize entry = 0. This is the minimum line length.
- For a line length of 32 bytes: $Log_2(32) = 5$, LineSize entry = 1.

Note

The C++ 17 specification has two defined parameters relating to the granularity of memory that does not interfere. For generic software and tools, Arm will set the hardware_destructive_interference_size parameter to 256 bytes and the hardware_constructive_interference_size parameter to 64 bytes.

Accessing CCSIDR_EL1

If CSSELR_EL1.{TnD, Level, InD} is programmed to a cache level that is not implemented, then on a read of the CCSIDR_EL1 the behavior is CONSTRAINED UNPREDICTABLE, and can be one of the following:

- The CCSIDR_EL1 read is treated as NOP.
- The CCSIDR_EL1 read is UNDEFINED. If FEAT_IDST is implemented, this is permitted to be reported with EC syndrome value `0x18`.
- The CCSIDR_EL1 read returns an UNKNOWN value.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, CCSIDR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b001	0b0000	0b0000	0b000

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TID2 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = CCSIDR_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = CCSIDR_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.22 CLIDR_EL1, Cache Level ID Register

The CLIDR_EL1 characteristics are:

Purpose

Identifies the type of cache, or caches, that are implemented at each level and can be managed using the architected cache maintenance instructions that operate by set/way, up to a maximum of seven levels. Also identifies the Level of Coherence (LoC) and Level of Unification (LoU) for the cache hierarchy.

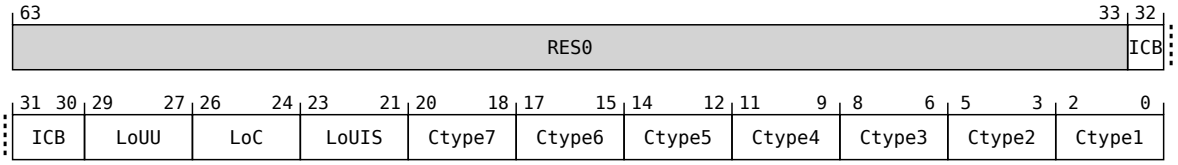
Configuration

There are no configuration notes.

Attributes

CLIDR_EL1 is a 64-bit register.

Field descriptions



Bits [63:33]

Reserved, RES0.

ICB, bits [32:30]

Inner cache boundary. This field indicates the boundary for caching Inner Cacheable memory regions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

ICB	Meaning
0b000	Not disclosed by this mechanism.
0b001	L1 cache is the highest Inner Cacheable level.
0b010	L2 cache is the highest Inner Cacheable level.
0b011	L3 cache is the highest Inner Cacheable level.
0b100	L4 cache is the highest Inner Cacheable level.
0b101	L5 cache is the highest Inner Cacheable level.
0b110	L6 cache is the highest Inner Cacheable level.
0b111	L7 cache is the highest Inner Cacheable level.

Access to this field is RO.

LoUU, bits [29:27]

Level of Unification Uniprocessor for the cache hierarchy.

For a description of the values of this field, see Terminology for Clean, Invalidate, and Clean and Invalidate instructions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

This field does not describe the requirements for instruction cache invalidation. See CTR_EL0.DIC.

Note

When FEAT_S2FWB is implemented, the architecture requires that this field is zero so that no levels of data cache need to be cleaned in order to manage coherency with instruction fetches.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

LoC, bits [26:24]

Level of Coherence for the cache hierarchy.

For a description of the values of this field, see Terminology for Clean, Invalidate, and Clean and Invalidate instructions.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

LoUIS, bits [23:21]

Level of Unification Inner Shareable for the cache hierarchy.

For a description of the values of this field, see Terminology for Clean, Invalidate, and Clean and Invalidate instructions.

Note

This field does not describe the requirements for instruction cache invalidation. See CTR_EL0.DIC.

Note

When FEAT_S2FWB is implemented, the architecture requires that this field is zero so that no levels of data cache need to be cleaned in order to manage coherency with instruction fetches.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Ctype<n> , bits [3(n-1)+2:3(n-1)], for n = 7 to 1

Cache Type fields. Indicate the type of cache that is implemented and can be managed using the architected cache maintenance instructions that operate by set/way at each level, from Level 1 up to a maximum of seven levels of cache hierarchy. Possible values of each field are:

Ctype<n>	Meaning
0b000	No cache.
0b001	Instruction cache only.
0b010	Data cache only.
0b011	Separate instruction and data caches.
0b100	Unified cache.

All other values are reserved.

If software reads the Cache Type fields from Ctype1 upwards, once it has seen a value of 0b000, no caches that can be managed using the architected cache maintenance instructions that operate by set/way exist at further-out levels of the hierarchy. So, for example, if Ctype3 is the first Cache Type field with a value of 0b000, the values of Ctype4 to Ctype7 must be ignored.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing CLIDR_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, CLIDR_EL1`

op0	op1	CRn	CRm	op2
0b11	0b001	0b0000	0b0000	0b001

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TID2 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = CLIDR_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = CLIDR_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.23 CONTEXTIDR_EL1, Context ID Register (EL1)

The CONTEXTIDR_EL1 characteristics are:

Purpose

Identifies the current Process Identifier.

The value of the whole of this register is called the Context ID and is used by:

- The debug logic, for Linked and Unlinked Context ID matching.
- The trace logic, to identify the current process.

The significance of this register is for debug and trace use only.

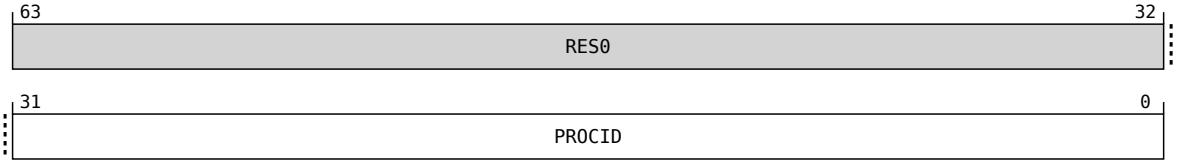
Configuration

There are no configuration notes.

Attributes

CONTEXTIDR_EL1 is a 64-bit register.

Field descriptions



Bits [63:32]

Reserved, RES0.

PROCID, bits [31:0]

Process Identifier. This field must be programmed with a unique value that identifies the current process.

Note

In AArch64 state, CONTEXTIDR_EL1 is independent of the ASID, and for the EL1&0 translation regime either TTBR0_EL1 or TTBR1_EL1 holds the ASID.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CONTEXTIDR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, CONTEXTIDR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1101	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TRVM == '1' then
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = CONTEXTIDR_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = CONTEXTIDR_EL1;
```

MSR CONTEXTIDR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1101	0b0000	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        CONTEXTIDR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    CONTEXTIDR_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.24 CONTEXTIDR_EL2, Context ID Register (EL2)

The CONTEXTIDR_EL2 characteristics are:

Purpose

Identifies the current Process Identifier for EL2.

The value of the whole of this register is called the Context ID and is used by:

- The debug logic, for Linked and Unlinked Context ID matching.
- The trace logic, to identify the current process.

The significance of this register is for debug and trace use only.

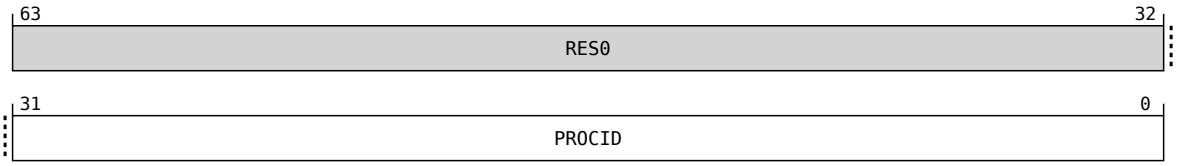
Configuration

This register is present only when FEAT_Debugv8p1 is implemented. Otherwise, direct accesses to CONTEXTIDR_EL2 are UNDEFINED.

Attributes

CONTEXTIDR_EL2 is a 64-bit register.

Field descriptions



Bits [63:32]

Reserved, RES0.

PROCID, bits [31:0]

Process Identifier. This field must be programmed with a unique value that identifies the current process.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CONTEXTIDR_EL2

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, CONTEXTIDR_EL2`

op0	op1	CRn	CRm	op2
0b11	0b100	0b1101	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_Debugv8p1) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = CONTEXTIDR_EL2;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR CONTEXTIDR_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1101	0b0000	0b001

```
if !IsFeatureImplemented(FEAT_Debugv8p1) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    CONTEXTIDR_EL2 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.25 CPACR_EL1, Architectural Feature Access Control Register

The CPACR_EL1 characteristics are:

Purpose

Controls access to trace, and Advanced SIMD and floating-point functionality.

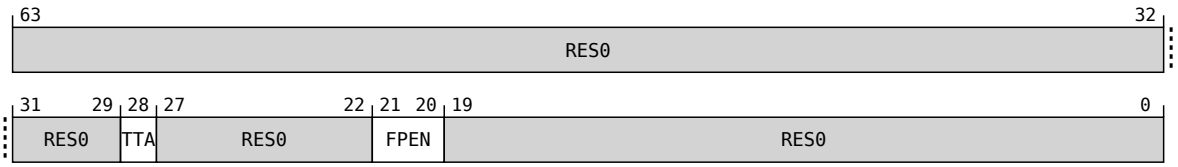
Configuration

There are no configuration notes.

Attributes

CPACR_EL1 is a 64-bit register.

Field descriptions



Bits [63:29]

Reserved, RES0.

TTA, bit [28]

When System register access to the trace unit registers is implemented:

Traps EL0 and EL1 System register accesses to all implemented trace registers from both Execution states to EL1, or to EL2 when it is implemented and enabled in the current Security state and HCR_EL2.TGE is 1, as follows:

- In AArch64 state, accesses to trace registers are trapped, reported using EC syndrome value `0x18`.

TTA	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	This control causes EL0 and EL1 System register accesses to all implemented trace registers to be trapped.

Note

- The ETMv4 architecture does not permit EL0 to access the trace registers. If the trace unit implements FEAT_ETMv4, EL0 accesses to the trace registers are UNDEFINED, and any resulting exception is higher priority than an exception that would be generated because the value of CPACR_EL1.TTA is 1.
- The Armv8-A architecture does not provide traps on trace register accesses through the optional memory-mapped interface.

System register accesses to the trace registers can have side-effects. When a System register access is trapped, any side-effects that are normally associated with the access do not occur before the exception is taken.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bits [27:22]

Reserved, RES0.

FPEN, bits [21:20]

Traps execution at EL1 and EL0 of instructions that access the Advanced SIMD and floating-point registers from both Execution states to EL1, reported using EC syndrome value 0x07, or to EL2 reported using EC syndrome value 0x00 when EL2 is implemented and enabled in the current Security state and HCR_EL2.TGE is 1, as follows:

- In AArch64 state, accesses to FPCR, FPSR, and any of the SIMD and floating-point registers V0-V31, including their views as D0-D31 registers or S0-31 registers.

FPEN	Meaning
0b00	This control causes execution of these instructions at EL1 and EL0 to be trapped.
0b01	This control causes execution of these instructions at EL0 to be trapped, but does not cause execution of any instructions at EL1 to be trapped.
0b10	This control causes execution of these instructions at EL1 and EL0 to be trapped.
0b11	This control does not cause execution of any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [19:0]

Reserved, RES0.

Accessing CPACR_EL1

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, CPACR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if CPTR_EL2.TCPAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = CPACR_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = CPACR_EL1;
```

MSR CPACR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b010

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if CPTR_EL2.TCPAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        CPACR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    CPACR_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.26 CPTR_EL2, Architectural Feature Trap Register (EL2)

The CPTR_EL2 characteristics are:

Purpose

Controls trapping to EL2 of accesses to CPACR, CPACR_EL1, trace, Activity Monitor, and Advanced SIMD and floating-point functionality.

Configuration

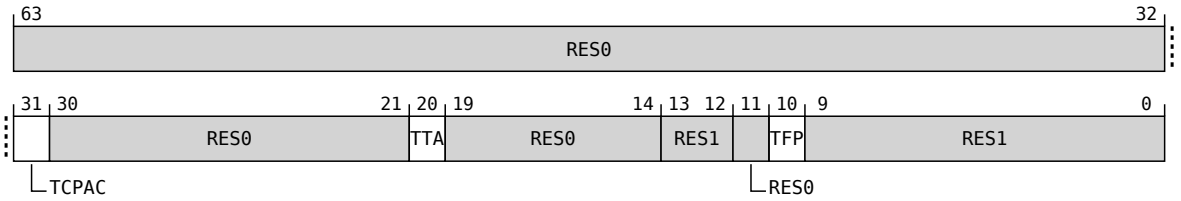
There are no configuration notes.

Attributes

CPTR_EL2 is a 64-bit register.

Field descriptions

!ELIsInHost(EL2)



This format applies in all Armv8.0 implementations.

Bits [63:32]

Reserved, RES0.

TCPAC, bit [31]

In AArch64 state, traps accesses to CPACR_EL1 from EL1 to EL2, when EL2 is enabled in the current Security state. The exception is reported using EC syndrome value `0x18`.

TCPAC	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	EL1 accesses to the following registers are trapped to EL2, when EL2 is enabled in the current Security state: • CPACR_EL1.

When HCR_EL2.TGE is 1, this control does not cause any instructions to be trapped.

Note

CPACR_EL1 and CPACR are not accessible at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [30:21]

Reserved, RES0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TTA, bit [20]

When System register access to the trace unit registers is implemented:

Traps System register accesses to all implemented trace registers from both Execution states to EL2, when EL2 is enabled in the current Security state, as follows:

- In AArch64 state, accesses to trace registers with op0=2, op1=1, and CRn<0b1000 are trapped to EL2, reported using EC syndrome value 0x18.

TTA	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	Any attempt at EL0, EL1, or EL2, to execute a System register access to an implemented trace register is trapped to EL2, when EL2 is enabled in the current Security state, unless it is trapped by one of the following controls: • CPACR_EL1.TTA.

Note

- FEAT_ETMv4 and FEAT_ETE do not permit EL0 to access the trace registers. EL0 accesses to the trace System registers are UNDEFINED, and any resulting exception is higher priority than an exception that would be generated because the value of CPTR_EL2.TTA is 1.
- EL2 does not provide traps on trace register accesses through the optional memory-mapped interface.

System register accesses to the trace registers can have side-effects. When a System register access is trapped, any side-effects that are normally associated with the access do not occur before the exception is taken.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [19:14]

Reserved, RES0.

Bits [13:12]

Reserved, RES1.

Bit [11]

Reserved, RES0.

TFP, bit [10]

Traps execution of instructions which access the Advanced SIMD and floating-point functionality, from both Execution states to EL2, when EL2 is enabled in the current Security state, as follows:

- In AArch64 state, accesses to the following registers are trapped to EL2, reported using EC syndrome value 0x07:
  - FPCR, FPSR, and any of the SIMD and floating-point registers V0-V31, including their views as D0-D31 registers or S0-31 registers.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TFP	Meaning
0b0	This control does not cause execution of any instructions to be trapped.
0b1	This control causes execution of these instructions at EL2, EL1, and EL0 to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [9:0]

Reserved, RES1.

Accessing CPTR_EL2

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, CPTR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = CPTR_EL2;
```

MSR CPTR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    CPTR_EL2 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.27 CSSELR_EL1, Cache Size Selection Register

The CSSELR_EL1 characteristics are:

Purpose

Selects the current Cache Size ID Register, CCSIDR_EL1, by specifying the required cache level and the cache type (either instruction or data cache).

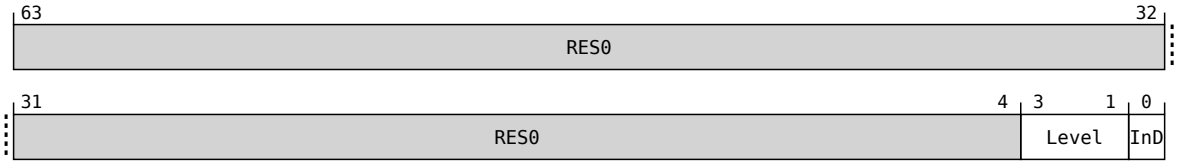
Configuration

There are no configuration notes.

Attributes

CSSELR_EL1 is a 64-bit register.

Field descriptions



Bits [63:4]

Reserved, RES0.

Level, bits [3:1]

Cache level of required cache.

Level	Meaning
0b000	Level 1 cache.
0b001	Level 2 cache.
0b010	Level 3 cache.
0b011	Level 4 cache.
0b100	Level 5 cache.
0b101	Level 6 cache.
0b110	Level 7 cache.

All other values are reserved.

If CSSELR_EL1.{Level, InD} is programmed to a cache level that is not implemented, then the value for this field on a read of CSSELR_EL1 is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

InD, bit [0]

Instruction not Data bit.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

InD	Meaning
0b0	Data or unified cache.
0b1	Instruction cache.

If CSSELR_EL1.{Level, InD} is programmed to a cache level that is not implemented, then a read of CSSELR_EL1 is CONSTRAINED UNPREDICTABLE, and returns UNKNOWN values for CSSELR_EL1.{Level, InD}.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CSSELR_EL1

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, CSSELR_EL1

op0	op1	CRn	CRm	op2
0b11	0b010	0b0000	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TID2 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = CSSELR_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = CSSELR_EL1;
```

MSR CSSELR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b010	0b0000	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TID2 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        CSSELR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    CSSELR_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.28 CTR_EL0, Cache Type Register

The CTR_EL0 characteristics are:

Purpose

Provides information about the architecture of the caches.

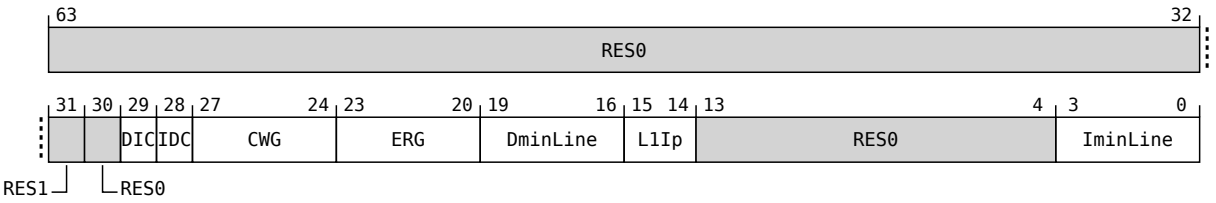
Configuration

There are no configuration notes.

Attributes

CTR_EL0 is a 64-bit register.

Field descriptions



Bits [63:32]

Reserved, RES0.

Bit [31]

Reserved, RES1.

Bit [30]

Reserved, RES0.

DIC, bit [29]

Instruction cache invalidation requirements for data to instruction coherence.

The value of this field is an IMPLEMENTATION DEFINED choice of:

DIC	Meaning
0b0	Instruction cache invalidation to the Point of Unification is required for data to instruction coherence.
0b1	Instruction cache invalidation to the Point of Unification is not required for data to instruction coherence.

All PEs in the same Inner Shareable shareability domain must have a common value of this field.

Note

If CTR_EL0.DIC is 1, instruction cache maintenance is not required after overwriting instructions, including for different VA aliases of the affected Locations, regardless of the value of CTR_EL0.L1Ip.

Access to this field is RO.

IDC, bit [28]

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Data cache clean requirements for instruction to data coherence. The meaning of this bit is:

The value of this field is an IMPLEMENTATION DEFINED choice of:

IDC	Meaning
0b0	Data cache clean to the Point of Unification is required for instruction to data coherence, unless CLIDR_EL1.LoC == 0b000 or (CLIDR_EL1.LoUIS == 0b000 and CLIDR_EL1.LoUU == 0b000).
0b1	Data cache clean to the Point of Unification is not required for instruction to data coherence.

If CTR_EL0.DIC is 1 then the value reported in this field must also be 1.

The Effective value of IDC is 1 if any of the following are true:

- CTR_EL0.IDC == 1.
- CLIDR_EL1.LoC == 0b000.
- CLIDR_EL1.LoUIS == 0b000 and CLIDR_EL1.LoUU == 0b000.

All PEs in the same Inner Shareable shareability domain must have a common Effective value of IDC.

Access to this field is RO.

CWG, bits [27:24]

Cache writeback granule. Log_2 of the number of words of the maximum size of memory that can be overwritten as a result of the eviction of a cache entry that has had a memory location in it modified.

A value of 0b0000 indicates that this register does not provide Cache writeback granule information and either:

- The architectural maximum of 512 words (2KB) must be assumed.
- The Cache writeback granule can be determined from maximum cache line size encoded in the Cache Size ID Registers.

Values greater than 0b1001 are reserved.

Arm recommends that an implementation that does not support cache write-back implements this field as 0b0001. This applies, for example, to an implementation that supports only write-through caches.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

ERG, bits [23:20]

Exclusives reservation granule. Log_2 of the number of words of the maximum size of the reservation granule that has been implemented for the Load-Exclusive and Store-Exclusive instructions.

The use of the value 0b0000 is deprecated.

The value 0b0001 and values greater than 0b1001 are reserved.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

DminLine, bits [19:16]

Log_2 of the number of words in the smallest cache line of all the data caches and unified caches that are controlled by the PE.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

L1Ip, bits [15:14]

Level 1 instruction cache policy. Indicates the indexing and tagging policy for the L1 instruction cache.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The value of this field is an IMPLEMENTATION DEFINED choice of:

L1Ip	Meaning
0b00	Reserved.
0b01	ASID-tagged Virtual Index, Virtual Tag (AIVIVT).
0b10	Virtual Index, Physical Tag (VIPT).
0b11	Physical Index, Physical Tag (PIPT).

From Armv8.0, the value 0b01 is reserved.

Access to this field is RO.

Bits [13:4]

Reserved, RES0.

IminLine, bits [3:0]

Log_2 of the number of words in the smallest cache line of all the instruction caches that are controlled by the PE.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing CTR_EL0

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, CTR_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b0000	0b0000	0b001

```
if PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && SCTLR_EL1.UCT == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif !ELIsInHost(EL0) && HCR_EL2.TID2 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = CTR_EL0;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TID2 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = CTR_EL0;
elsif PSTATE.EL == EL2 then
    X[t, 64] = CTR_EL0;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.29 DCZID_EL0, Data Cache Zero ID Register

The DCZID_EL0 characteristics are:

Purpose

Indicates the block size that is written with byte values of 0 by the DC ZVA (Data Cache Zero by Address) System instruction.

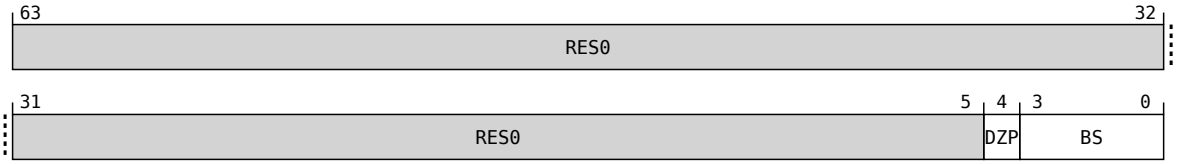
Configuration

There are no configuration notes.

Attributes

DCZID_EL0 is a 64-bit register.

Field descriptions



Bits [63:5]

Reserved, RES0.

DZP, bit [4]

Data Zero Prohibited. This field indicates whether use of DC ZVA instructions is permitted or prohibited.

DZP	Meaning
0b0	Instructions are permitted.
0b1	Instructions are prohibited.

The value read from this field is governed by the current Exception level and the values of the following fields:

- The Effective value of HCR_EL2.TDZ.
- When the Effective value of HCR_EL2.{E2H, TGE} != '11', SCTLR_EL1.DZE.

BS, bits [3:0]

Log_2 of the block size in words written by a DC ZVA instruction. The maximum size supported is 2KB, indicated by value 0b1001.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing DCZID_EL0

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, DCZID_EL0
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b011	0b0000	0b0000	0b111

```
if PSTATE.EL == EL0 then
    X[t, 64] = DCZID_EL0;
elsif PSTATE.EL == EL1 then
    X[t, 64] = DCZID_EL0;
elsif PSTATE.EL == EL2 then
    X[t, 64] = DCZID_EL0;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.30 ESR_EL1, Exception Syndrome Register (EL1)

The ESR_EL1 characteristics are:

Purpose

Holds syndrome information for an exception taken to EL1.

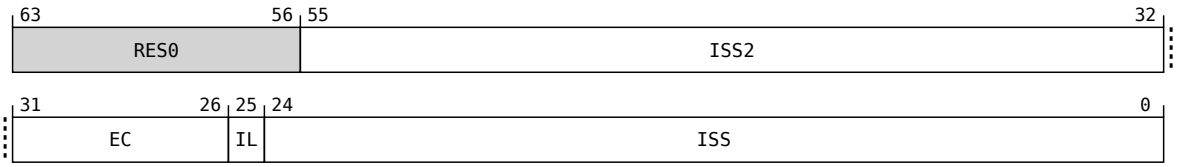
Configuration

There are no configuration notes.

Attributes

ESR_EL1 is a 64-bit register.

Field descriptions



ESR_EL1 is made UNKNOWN as a result of an exception return from EL1.

When an UNPREDICTABLE instruction is treated as UNDEFINED, and the exception is taken to EL1, the value of ESR_EL1 is UNKNOWN. The value written to ESR_EL1 must be consistent with a value that could be created as a result of an exception from the same Exception level that generated the exception as a result of a situation that is not UNPREDICTABLE at that Exception level, in order to avoid the possibility of a privilege violation.

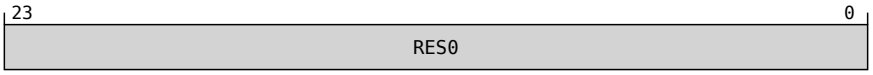
Bits [63:56]

Reserved, RES0.

ISS2, bits [55:32]

ISS2 encoding for an exception, the bit assignments are:

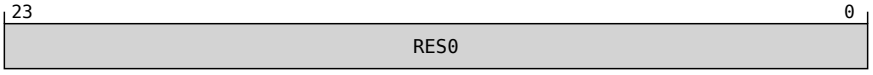
ISS2 encoding for an exception from a Data Abort



Bits [23:0]

Reserved, RES0.

ISS2 encoding for an exception from an Instruction Abort

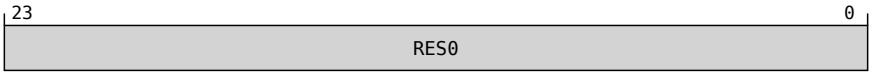


Bits [23:0]

Reserved, RES0.

ISS2 encoding for an exception from a Watchpoint exception

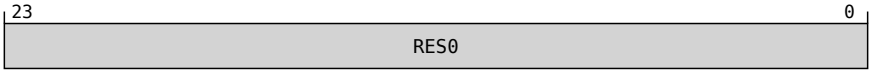
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential



Bits [23:0]

Reserved, RES0.

ISS2 encoding for all other exceptions



Bits [23:0]

Reserved, RES0.

EC, bits [31:26]

Exception Class. Indicates the reason for the exception that this register holds information about.

For each EC value, the table references a subsection that gives information about:

- The cause of the exception, for example the configuration required to enable the trap.
- The encoding of the associated ISS.

Possible values of the EC field are:

EC	Meaning	ISS	ISS2	Applies when
0b000000	Unknown reason.	ISS encoding for exceptions with an unknown reason	ISS2 encoding for all other exceptions	
0b000001	Trapped WF* instruction execution. Conditional WF* instructions that fail their condition code check do not cause an exception.	ISS encoding for an exception from a WF* instruction	ISS2 encoding for all other exceptions	
0b000111	Access to Advanced SIMD or floating-point functionality trapped by CPACR_EL1.FPEN, CPTR_EL2.FPEN, CPTR_EL2.TFP, or CPTR_EL3.TFP control. Excludes exceptions resulting from CPACR_EL1 when the value of HCR_EL2.TGE is 1, or because Advanced SIMD and floating-point are not implemented. These are reported with EC value 0b000000.	ISS encoding for an exception from an access to a register or instruction resulting from the FPEN and TFP traps	ISS2 encoding for all other exceptions	
0b001001	Trapped use of a Pointer authentication instruction.	ISS encoding for an exception from a trapped Pointer Authentication instruction	ISS2 encoding for all other exceptions	FEAT_PAuth is implemented
0b001110	Illegal Execution state.	ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault	ISS2 encoding for all other exceptions	

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EC	Meaning	ISS	ISS2	Applies when
0b010101	SVC instruction execution in AArch64 state.	ISS encoding for an exception from HVC or SVC instruction execution	ISS2 encoding for all other exceptions	
0b011000	Trapped MSR, MRS or System instruction execution in AArch64 state, that is not reported using EC values 0b000000, 0b000001, or 0b000111. This includes all instructions that cause exceptions that are part of the encoding space defined in 'System instruction class encoding overview', except for those exceptions reported using EC values 0b000000, 0b000001, or 0b000111.	ISS encoding for an exception from MSR, MRS, or System instruction execution in AArch64 state	ISS2 encoding for all other exceptions	
0b011100	Exception from a PAC Fail	ISS encoding for a PAC Fail exception	ISS2 encoding for all other exceptions	FEAT_FPAC is implemented
0b100000	Instruction Abort from a lower Exception level. Used for MMU and MPU faults generated by instruction accesses and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug-related exceptions.	ISS encoding for an exception from an Instruction Abort	ISS2 encoding for an exception from an Instruction Abort	
0b100001	Instruction Abort taken without a change in Exception level. Used for MMU and MPU faults generated by instruction accesses and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug-related exceptions.	ISS encoding for an exception from an Instruction Abort	ISS2 encoding for an exception from an Instruction Abort	
0b100010	PC alignment fault exception.	ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault	ISS2 encoding for all other exceptions	
0b100100	Data Abort exception from a lower Exception level. Used for MMU and MPU faults generated by data accesses, alignment faults other than those caused by Stack Pointer misalignment, and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug-related exceptions.	ISS encoding for an exception from a Data Abort	ISS2 encoding for an exception from a Data Abort	

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EC	Meaning	ISS	ISS2	Applies when
0b100101	Data Abort exception taken without a change in Exception level. Used for MMU and MPU faults generated by instruction accesses and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug-related exceptions.	ISS encoding for an exception from a Data Abort	ISS2 encoding for an exception from a Data Abort	
0b100110	SP alignment fault exception.	ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault	ISS2 encoding for all other exceptions	
0b101100	Trapped floating-point exception taken from AArch64 state. This EC value is valid if the implementation supports trapping of floating-point exceptions, otherwise it is reserved. Whether a floating-point implementation supports trapping of floating-point exceptions is IMPLEMENTATION DEFINED.	ISS encoding for an exception from a trapped floating-point exception	ISS2 encoding for all other exceptions	
0b101111	SError exception.	ISS encoding for an SError exception	ISS2 encoding for all other exceptions	
0b110000	Breakpoint exception from a lower Exception level.	ISS encoding for an exception from a Breakpoint or Vector Catch debug exception	ISS2 encoding for all other exceptions	
0b110001	Breakpoint exception taken without a change in Exception level.	ISS encoding for an exception from a Breakpoint or Vector Catch debug exception	ISS2 encoding for all other exceptions	
0b110010	Software Step exception from a lower Exception level.	ISS encoding for an exception from a Software Step exception	ISS2 encoding for all other exceptions	
0b110011	Software Step exception taken without a change in Exception level.	ISS encoding for an exception from a Software Step exception	ISS2 encoding for all other exceptions	
0b110100	Watchpoint exception from a lower Exception level.	ISS encoding for an exception from a Watchpoint exception	ISS2 encoding for an exception from a Watchpoint exception	
0b110101	Watchpoint exception taken without a change in Exception level.	ISS encoding for an exception from a Watchpoint exception	ISS2 encoding for an exception from a Watchpoint exception	

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b111100	BRK instruction execution in AArch64 state.	ISS encoding for an exception from execution of a Breakpoint instruction	ISS2 encoding for all other exceptions

All other EC values are reserved by Arm, and:

- Unused values in the range 0b000000 - 0b101100 (0x00 - 0x2C) are reserved for future use for synchronous exceptions.
- Unused values in the range 0b101101 - 0b111111 (0x2D - 0x3F) are reserved for future use, and might be used for synchronous or asynchronous exceptions.

The effect of programming this field to a reserved value is that behavior is CONSTRAINED UNPREDICTABLE.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IL, bit [25]

Instruction Length for synchronous exceptions. Possible values of this bit are:

IL	Meaning
0b0	16-bit instruction trapped.

0b1	32-bit instruction trapped. This value is also used when the exception is one of the following:
	• An SError exception.
	• An Instruction Abort exception.
	• A PC alignment fault exception.
	• An SP alignment fault exception.
	• A Data Abort exception for which the value of the ISV bit is 0.
	• An Illegal Execution state exception.
	• Any debug exception except for Breakpoint instruction exceptions. For Breakpoint instruction exceptions, this bit has its standard meaning:
	  • 0b0: 16-bit T32 BKPT instruction.
	  • 0b1: 32-bit A32 BKPT instruction or A64 BRK instruction.
	• An exception reported using EC value 0b000000.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

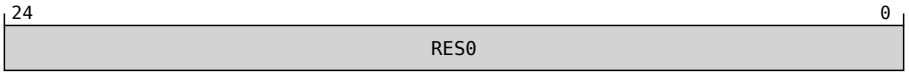
ISS, bits [24:0]

Instruction Specific Syndrome. Architecturally, this field can be defined independently for each defined Exception class. However, in practice, some ISS encodings are used for more than one Exception class.

Typically, an ISS encoding has a number of subfields. When an ISS subfield holds a register number, the value returned in that field is the AArch64 view of the register number.

ISS encoding for exceptions with an unknown reason

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential



Bits [24:0]

Reserved, RES0.

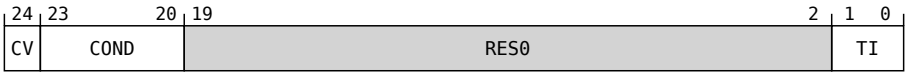
Additional Information for ISS encoding for exceptions with an unknown reason

When an exception is reported using this EC value, the IL field is set to 1.

This EC value is used for all exceptions that are not covered by any other EC value. This includes exceptions that are generated in the following situations:

- The attempted execution of an instruction bit pattern that has no allocated instruction or that is not accessible at the current Exception level and Security state, including:
  - A read access using a System register pattern that is not allocated for reads or that does not permit reads at the current Exception level and Security state.
  - A write access using a System register pattern that is not allocated for writes or that does not permit writes at the current Exception level and Security state.
  - Instruction encodings that are unallocated.
  - Instruction encodings for instructions or System registers that are not implemented in the implementation.
- In Debug state, the attempted execution of an instruction bit pattern that is not accessible in Debug state.
- In Non-debug state, the attempted execution of an instruction bit pattern that is not accessible in Non-debug state.
- In an implementation that does not include Advanced SIMD and floating-point functionality, an attempted access to Advanced SIMD or floating-point functionality under conditions where that access would be permitted if that functionality was present. This includes the attempted execution of an Advanced SIMD or floating-point instruction, and attempted accesses to Advanced SIMD and floating-point System registers.
- An exception generated because of the value of one of the SCTLR_EL1.{ITD, SED, CP15BEN} control bits.
- Attempted execution of:
  - An HVC instruction when disabled by HCR_EL2.HCD.
  - An SMC instruction when disabled by HCR_EL2.TSC.
  - An HLT instruction when disabled by EDSCR.HDE.
- Attempted execution of an MSR or MRS instruction to access SP_EL0 when the value of SPSel.SP is 0.
- Attempted execution of an MSR or MRS instruction using a _EL12 register name.
- Attempted execution, in Debug state, of:
  - A DCPS1 instruction when the value of HCR_EL2.TGE is 1.
  - A DCPS2 instruction from EL1 or EL0 when EL2 is disabled or not implemented in the current Security state.
  - A DCPS3 instruction when the value of EDSCR.SDD is 1.
- In Debug state when the value of EDSCR.SDD is 1, the attempted execution at EL1 or EL0 of an instruction that is configured to trap to EL2.
- An exception that is taken to EL2 because the value of HCR_EL2.TGE is 1. If the value of HCR_EL2.TGE is 0, this exception is reported using an ESR_EL1.EC value of `0b000111`.

ISS encoding for an exception from a WF* instruction



CV, bit [24]

Condition code valid.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CV	Meaning
0b0	The COND field is not valid.
0b1	The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

For exceptions taken from AArch64, this field is set to 0b1110.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [19:2]

Reserved, RES0.

TI, bits [1:0]

Trapped instruction. Possible values of this bit are:

TI	Meaning
0b00	WFI trapped.
0b01	WFE trapped.

The reset behavior of this field is:

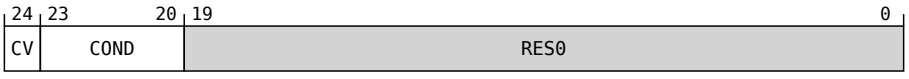
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from a WF* instruction

The following fields describe configuration settings for generating this exception:

- SCTLR_EL1.{nTWE, nTWI}.
- SCTLR_EL2.{nTWE, nTWI}.
- HCR_EL2.{TWE, TWI}.

ISS encoding for an exception from an access to a register or instruction resulting from the FPEN and TFP traps



The accesses covered by this trap include:

- Execution of Advanced SIMD and floating-point instructions.
- Accesses to the Advanced SIMD and floating-point System registers.

For an implementation that does not include support for Advanced SIMD and floating-point, the exception is reported using the EC value 0b000000.

CV, bit [24]

Condition code valid.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CV	Meaning
0b0	The COND field is not valid.
0b1	The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

For exceptions taken from AArch64, this field is set to 0b1110.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [19:0]

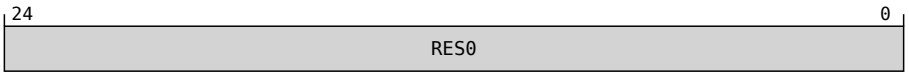
Reserved, RES0.

Additional Information for ISS encoding for an exception from an access to a register or instruction resulting from the FPEN and TFP traps

The following fields describe the configuration settings for the traps that are reported using EC value 0b000111:

- CPACR_EL1.FPEN, for accesses to SIMD and floating-point registers trapped to EL1.
- CPTR_EL2.FPEN and CPTR_EL2.TFP, for accesses to SIMD and floating-point registers trapped to EL2.

ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault



Bits [24:0]

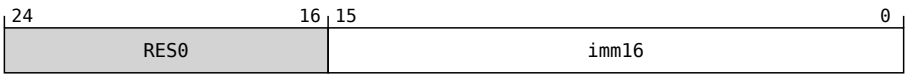
Reserved, RES0.

Additional Information for ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault

There are no configuration settings for generating Illegal Execution state exceptions and PC alignment fault exceptions. For more information about PC alignment fault exceptions, see 'PC alignment checking'.

'SP alignment checking' describes the configuration settings for generating SP alignment fault exceptions.

ISS encoding for an exception from HVC or SVC instruction execution



Bits [24:16]

Reserved, RES0.

imm16, bits [15:0]

The value of the immediate field from the HVC or SVC instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For an HVC instruction, and for an A64 SVC instruction, this is the value of the imm16 field of the issued instruction.

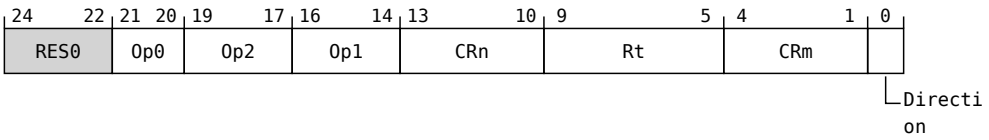
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from HVC or SVC instruction execution

For A64 instructions, see 'SVC' and 'HVC'.

ISS encoding for an exception from MSR, MRS, or System instruction execution in AArch64 state



Bits [24:22]

Reserved, RES0.

Op0, bits [21:20]

The Op0 value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Op2, bits [19:17]

The Op2 value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Op1, bits [16:14]

The Op1 value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CRn, bits [13:10]

The CRn value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Rt, bits [9:5]

The Rt value from the issued instruction, the general-purpose register used for the transfer.

For system instructions which require that the opcode Rt field is set to `0b11111`, but where the trapped instruction has a different value of Rt, an implementation is permitted to return the value `0b11111`, instead of the value of Rt from the trapped instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CRm, bits [4:1]

The CRm value from the issued instruction.

The reset behavior of this field is:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Direction, bit [0]

Indicates the direction of the trapped instruction.

Direction	Meaning
0b0	Write access, including MSR instructions.
0b1	Read access, including MRS instructions.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from MSR, MRS, or System instruction execution in AArch64 state

For exceptions caused by System instructions, see 'System instructions' subsection of 'Branches, exception generating and System instructions' for the encoding values returned by an instruction.

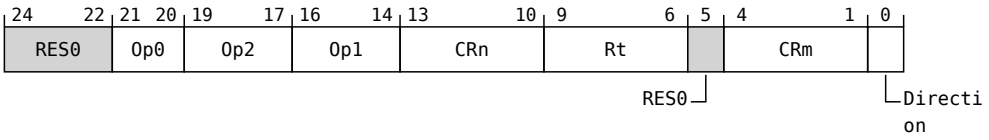
The following fields describe configuration settings for generating the exception that is reported using EC value 0b011000:

- SCTLR_EL1.UCI, for execution of cache maintenance instructions using AArch64 state, execution is trapped to EL1 or EL2.
- SCTLR_EL1.UCT, for accesses to CTR_EL0 using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- SCTLR_EL1.DZE, for execution of DC ZVA instructions using AArch64 state, execution is trapped to EL1 or EL2.
- SCTLR_EL1.UMA, for accesses to the PSTATE interrupt masks using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- CPACR_EL1.TTA, for accesses to the trace registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- MDSCR_EL1.TDCC, for accesses to the Debug Communications Channel (DCC) registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- CNTKCTL_EL1.{EL0PTEN, EL0VTEN, EL0PCTEN, EL0VCTEN} accesses to the Generic Timer registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- PMUSERENR_EL0, for accesses to the Performance Monitor registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- AMUSERENR_EL0.EN, for accesses to Activity Monitors registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- HCR_EL2.{TRVM, TVM}, for accesses to virtual memory control registers using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.TDZ, for execution of DC ZVA instructions using AArch64 state, execution is trapped to EL2.
- HCR_EL2.TTLB, for execution of TLB maintenance instructions using AArch64 state, execution is trapped to EL2.
- HCR_EL2.{TSW, TPC, TPU}, for execution of cache maintenance instructions using AArch64 state, execution is trapped to EL2.
- HCR_EL2.TACR, for accesses to the Auxiliary Control Register, ACTLR_EL1, using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.TIDCP, for accesses to lockdown, DMA, and TCM operations using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.{TID1, TID2, TID3}, for accesses to ID group 1, ID group 2 or ID group 3 registers, using AArch64 state, MSR or MRS access trapped to EL2.
- CPTR_EL2.TCPAC, for accesses to CPACR_EL1, using AArch64 state, MSR or MRS access trapped to EL2.
- CPTR_EL2.TTA, for accesses to the trace registers, using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.TTRF, for accesses to the trace filter control register, TRFCR_EL1, using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.TDRA, for accesses to Debug ROM registers, using AArch64 state, MSR or MRS access trapped to EL2.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- MDCR_EL2.TDOSA, for accesses to powerdown debug registers using AArch64 state, MSR or MRS access trapped to EL2.
- CNTHCTL_EL2.{EL1PCEN, EL1PCTEN}, for accesses to the Generic Timer registers using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.TDA, for accesses to debug registers using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.{TPM, TPMCR}, for accesses to Performance Monitor registers, using AArch64 state, MSR or MRS access trapped to EL2.
- CPTR_EL2.TAM, for accesses to Activity Monitors registers, using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.APK, for accesses to Pointer authentication key registers, using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.{NV, NV1}, for Nested virtualization register access, using AArch64 state, MSR or MRS access, trapped to EL2.
- HCR_EL2.AT, for execution of AT S1E* instructions, using AArch64 state, execution is trapped to EL2.
- HCR_EL2.{TERR, FIEN}, for accesses to RAS registers, using AArch64 state, MSR or MRS access, trapped to EL2.
- CPTR_EL3.TAM, for accesses to Activity Monitors registers, using AArch64 state, MSR or MRS access, trapped to EL3.

ISS encoding for an exception from MSRR, MRRS, or 128-bit System instruction execution in AArch64 state



Bits [24:22]

Reserved, RES0.

Op0, bits [21:20]

The Op0 value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Op2, bits [19:17]

The Op2 value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Op1, bits [16:14]

The Op1 value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CRn, bits [13:10]

The CRn value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Rt, bits [9:6]

The Rt value from the issued instruction, the general-purpose register used for the transfer.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

This value represents register pair of X[Rt:0], X[Rt:1].

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [5]

Reserved, RES0.

CRm, bits [4:1]

The CRm value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Direction, bit [0]

Indicates the direction of the trapped instruction.

Direction	Meaning
0b0	Write access, MSRR instructions.
0b1	Read access, MRRS instructions.

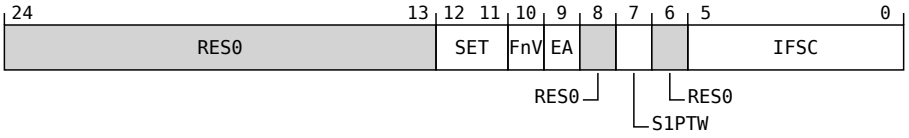
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from MSRR, MRRS, or 128-bit System instruction execution in AArch64 state

The following fields describe configuration settings for generating exceptions from an MSRR or MRRS access that are reported using EC value 0b010100:

ISS encoding for an exception from an Instruction Abort



Bits [24:13]

Reserved, RES0.

SET, bits [12:11]

When IsFeatureImplemented(FEAT_RAS) && ((IFSC == 0b010000 || IFSC IN {0b01001x}) || IFSC IN {0b0101xx}):

Synchronous Error Type. Describes the PE error state after taking the Instruction Abort exception.

SET	Meaning
0b00	Recoverable state (UER).
0b10	Uncontainable (UC).
0b11	Restartable state (UEO).

All other values are reserved.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Software can use this information to determine what recovery might be possible. Taking a synchronous External abort exception might result in a PE state that is not recoverable.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

FnV, bit [10]

FAR not Valid, for a synchronous External abort other than a synchronous External abort on a translation table walk.

FnV	Meaning
0b0	FAR is valid.
0b1	FAR is not valid, and holds an UNKNOWN value.

This field is valid only if the IFSC code is 0b010000. It is RES0 for all other aborts.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EA, bit [9]

External abort type. This bit can provide an IMPLEMENTATION DEFINED classification of External aborts.

For any abort other than an External abort this bit returns a value of 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [8]

Reserved, RES0.

S1PTW, bit [7]

For a stage 2 fault, indicates whether the fault was a stage 2 fault on an access made for a stage 1 translation table walk:

S1PTW	Meaning
0b0	Fault not on a stage 2 translation for a stage 1 translation table walk.
0b1	Fault on the stage 2 translation of an access for a stage 1 translation table walk.

For any abort other than a stage 2 fault this bit is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [6]

Reserved, RES0.

IFSC, bits [5:0]

Instruction Fault Status Code.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IFSC	Meaning	Applies when
0b000000	Address size fault, level 0 of translation or translation table base register.	
0b000001	Address size fault, level 1.	
0b000010	Address size fault, level 2.	
0b000011	Address size fault, level 3.	
0b000100	Translation fault, level 0.	
0b000101	Translation fault, level 1.	
0b000110	Translation fault, level 2.	
0b000111	Translation fault, level 3.	
0b001001	Access flag fault, level 1.	
0b001010	Access flag fault, level 2.	
0b001011	Access flag fault, level 3.	
0b001100	Permission fault, level 0.	
0b001101	Permission fault, level 1.	
0b001110	Permission fault, level 2.	
0b001111	Permission fault, level 3.	
0b010000	Synchronous External abort, not on translation table walk or hardware update of translation table.	
0b010100	Synchronous External abort on translation table walk or hardware update of translation table, level 0.	
0b010101	Synchronous External abort on translation table walk or hardware update of translation table, level 1.	
0b010110	Synchronous External abort on translation table walk or hardware update of translation table, level 2.	
0b010111	Synchronous External abort on translation table walk or hardware update of translation table, level 3.	
0b011000	Synchronous parity or ECC error on memory access, not on translation table walk.	FEAT_RAS is not implemented
0b011100	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 0.	FEAT_RAS is not implemented
0b011101	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 1.	FEAT_RAS is not implemented
0b011110	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 2.	FEAT_RAS is not implemented
0b011111	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 3.	FEAT_RAS is not implemented
0b110000	TLB conflict abort.	
0b110001	Unsupported atomic hardware update fault.	FEAT_HAFDBS is implemented

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

All other values are reserved.

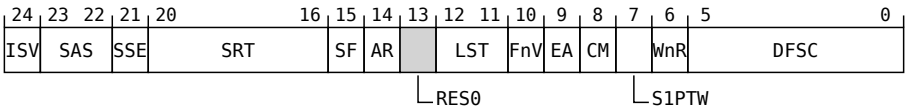
For more information about the lookup level associated with a fault, see 'The lookup level associated with MMU faults'.

If the S1PTW bit is set, then the level refers the level of the stage2 translation that is translating a stage 1 translation walk.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ISS encoding for an exception from a Data Abort



ISV, bit [24]

Instruction Syndrome Valid. Indicates whether the syndrome information in ISS[23:14] is valid.

ISV	Meaning
0b0	No valid instruction syndrome. ISS[23:14] are RES0.
0b1	ISS[23:14] hold a valid instruction syndrome.

This bit is 0 for all faults reported in ESR_EL2 except the following stage 2 aborts:

- AArch64 loads and stores of a single general-purpose register (including the register specified with 0b11111, including those with Acquire/Release semantics, but excluding Load Exclusive or Store Exclusive and excluding those with writeback).

For these stage 2 aborts, ISV is UNKNOWN if the exception was generated in Debug state in Memory access mode, and otherwise indicates whether ISS[23:14] hold a valid syndrome.

ISV is 0 for all faults reported in ESR_EL1 or ESR_EL3.

When FEAT_RAS is implemented, ISV is 0 for any synchronous External abort.

When FEAT_RAS is not implemented, it is IMPLEMENTATION DEFINED whether ISV is set to 1 or 0 on a synchronous External abort on a stage 2 translation table walk.

For ISS reporting, a stage 2 abort on a stage 1 translation table walk does not return a valid instruction syndrome, and therefore ISV is 0 for these aborts.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SAS, bits [23:22]

When ISV == '1':

Syndrome Access Size. Indicates the size of the access attempted by the faulting operation.

SAS	Meaning
0b00	Byte
0b01	Halfword
0b10	Word
0b11	Doubleword

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This field is UNKNOWN when the value of ISV is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SSE, bit [21]

When ISV == '1':

Syndrome Sign Extend. For a byte, halfword, or word load operation, indicates whether the data item must be sign extended.

SSE	Meaning
0b0	Sign-extension not required.
0b1	Data item must be sign-extended.

For all other operations, this field is 0.

This field is UNKNOWN when the value of ISV is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SRT, bits [20:16]

When ISV == '1':

Syndrome Register Transfer. The register number of the Wt/Xt/Rt operand of the faulting instruction.

This field is UNKNOWN when the value of ISV is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [15]

When ISV == '1':

SF, bit [15]

Sixty Four bit general-purpose register transfer. Width of the register accessed by the instruction is 64-bit.

SF	Meaning
0b0	Instruction loads/stores a 32-bit general-purpose register.
0b1	Instruction loads/stores a 64-bit general-purpose register.

Note

This field specifies the register width identified by the instruction, not the Execution state.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This field is UNKNOWN when the value of ISV is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

FnP, bit [15]

FAR not Precise.

FnP	Meaning
0b0	The FAR holds the faulting virtual address that generated the Data Abort.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

AR, bit [14]

When ISV == '1':

Acquire/Release.

AR	Meaning
0b0	Instruction did not have acquire/release semantics.
0b1	Instruction did have acquire/release semantics.

This field is UNKNOWN when the value of ISV is UNKNOWN.

For a load-acquire instruction that does not have acquire semantics as the result of the destination register being ZR, it is IMPLEMENTATION SPECIFIC whether this field is reported as 0 or 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [13]

Reserved, RES0.

Bits [12:11]

When (DFSC IN {0b00xxxx} || DFSC IN {0b10101x}) && !(DFSC IN {0b0000xx}):

LST, bits [12:11]

Load/Store Type. Used when a Translation fault, Access flag fault, or Permission fault generates a Data Abort.

LST	Meaning
0b00	The instruction that generated the Data Abort is not specified by this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When IsFeatureImplemented(FEAT_RAS) && ((DFSC == 0b010000 || DFSC IN {0b01001x}) || DFSC IN {0b0101xx}):

SET, bits [12:11]

Synchronous Error Type. Used when a synchronous External abort, not on a Translation table walk or hardware update of the Translation table, generated the Data Abort. Describes the PE error state after taking the Data Abort exception.

SET	Meaning
0b00	Recoverable state (UER).
0b10	Uncontainable (UC).
0b11	Restartable state (UEO).

Note

Software can use this information to determine what recovery might be possible. Taking a synchronous External abort exception might result in a PE state that is not recoverable.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

FnV, bit [10]

FAR not Valid, for a synchronous External abort other than a synchronous External abort on a translation table walk.

FnV	Meaning
0b0	FAR is valid.
0b1	FAR is not valid, and holds an UNKNOWN value.

This field is valid only if the DFSC code is 0b010000. It is RES0 for all other aborts.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EA, bit [9]

External abort type. This bit can provide an IMPLEMENTATION DEFINED classification of External aborts.

For any abort other than an External abort this bit returns a value of 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CM, bit [8]

Cache maintenance. Indicates whether the Data Abort came from a cache maintenance or address translation instruction:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CM	Meaning
0b0	The Data Abort was not generated by the execution of one of the System instructions identified in the description of value 1.
0b1	The Data Abort was generated by either the execution of a cache maintenance instruction or by a synchronous fault on the execution of an address translation instruction. The DC ZVA, DC GVA, and DC GZVA instructions are not classified as cache maintenance instructions, and therefore their execution cannot cause this field to be set to 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

S1PTW, bit [7]

For a stage 2 fault, indicates whether the fault was a stage 2 fault on an access made for a stage 1 translation table walk:

S1PTW	Meaning
0b0	Fault not on a stage 2 translation for a stage 1 translation table walk.
0b1	Fault on the stage 2 translation of an access for a stage 1 translation table walk.

For any abort other than a stage 2 fault this bit is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

WnR, bit [6]

Write not Read. Indicates whether a synchronous abort was caused by an instruction writing to a memory location, or by an instruction reading from a memory location.

WnR	Meaning
0b0	Abort caused by an instruction reading from a memory location.
0b1	Abort caused by an instruction writing to a memory location.

For faults on cache maintenance and address translation instructions, this bit always returns a value of 1.

For faults from an atomic instruction that both reads and writes from a memory location, this bit is set to 0 if a read of the address specified by the instruction would have generated the fault which is being reported, otherwise it is set to 1. The architecture permits, but does not require, a relaxation of this requirement such that for all stage 2 aborts on stage 1 translation table walks for atomic instructions, the WnR bit is always 0.

This field is UNKNOWN for:

- An External abort on an Atomic access.
- A fault reported using a DFSC value of 0b110101 or 0b110001, indicating an unsupported Exclusive or atomic access.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

DFSC, bits [5:0]

Data Fault Status Code.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DFSC	Meaning	Applies when
0b000000	Address size fault, level 0 of translation or translation table base register.	
0b000001	Address size fault, level 1.	
0b000010	Address size fault, level 2.	
0b000011	Address size fault, level 3.	
0b000100	Translation fault, level 0.	
0b000101	Translation fault, level 1.	
0b000110	Translation fault, level 2.	
0b000111	Translation fault, level 3.	
0b001001	Access flag fault, level 1.	
0b001010	Access flag fault, level 2.	
0b001011	Access flag fault, level 3.	
0b001100	Permission fault, level 0.	
0b001101	Permission fault, level 1.	
0b001110	Permission fault, level 2.	
0b001111	Permission fault, level 3.	
0b010000	Synchronous External abort, not on translation table walk or hardware update of translation table.	
0b010100	Synchronous External abort on translation table walk or hardware update of translation table, level 0.	
0b010101	Synchronous External abort on translation table walk or hardware update of translation table, level 1.	
0b010110	Synchronous External abort on translation table walk or hardware update of translation table, level 2.	
0b010111	Synchronous External abort on translation table walk or hardware update of translation table, level 3.	
0b011000	Synchronous parity or ECC error on memory access, not on translation table walk.	FEAT_RAS is not implemented
0b011100	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 0.	FEAT_RAS is not implemented
0b011101	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 1.	FEAT_RAS is not implemented
0b011110	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 2.	FEAT_RAS is not implemented
0b011111	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 3.	FEAT_RAS is not implemented
0b100001	Alignment fault.	
0b110000	TLB conflict abort.	
0b110001	Unsupported atomic hardware update fault.	FEAT_HAFDBS is implemented
0b110100	IMPLEMENTATION DEFINED fault (Lockdown).	

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b110101	IMPLEMENTATION DEFINED fault (Unsupported Exclusive or Atomic access).

All other values are reserved.

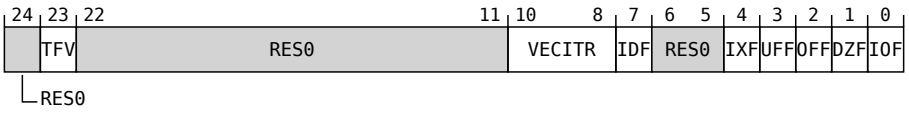
For more information about the lookup level associated with a fault, see 'The lookup level associated with MMU faults'.

If the S1PTW bit is set, then the level refers the level of the stage2 translation that is translating a stage 1 translation walk.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ISS encoding for an exception from a trapped floating-point exception



Bit [24]

Reserved, RES0.

TFV, bit [23]

Trapped Fault Valid bit. Indicates whether the IDF, IXF, UFF, OFF, DZF, and IOF bits hold valid information about trapped floating-point exceptions.

TFV	Meaning
0b0	The IDF, IXF, UFF, OFF, DZF, and IOF bits do not hold valid information about trapped floating-point exceptions and are UNKNOWN.
0b1	One or more floating-point exceptions occurred during an operation performed while executing the reported instruction. The IDF, IXF, UFF, OFF, DZF, and IOF bits indicate trapped floating-point exceptions that occurred. For more information, see 'Floating-point exceptions and exception traps'.

It is IMPLEMENTATION DEFINED whether this field is set to 0 on an exception generated by a trapped floating-point exception from an instruction that is performing floating-point operations on more than one lane of a vector.

Note

This is not a requirement. Implementations can set this field to 1 on a trapped floating-point exception from an instruction and return valid information in the {IDF, IXF, UFF, OFF, DZF, IOF} fields.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [22:11]

Reserved, RES0.

VECITR, bits [10:8]

For a trapped floating-point exception from an instruction executed in AArch64 state this field is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IDF, bit [7]

Input Denormal floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

IDF	Meaning
0b0	Input denormal floating-point exception has not occurred.
0b1	Input denormal floating-point exception occurred during execution of the reported instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [6:5]

Reserved, RES0.

IXF, bit [4]

Inexact floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

IXF	Meaning
0b0	Inexact floating-point exception has not occurred.
0b1	Inexact floating-point exception occurred during execution of the reported instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

UFF, bit [3]

Underflow floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

UFF	Meaning
0b0	Underflow floating-point exception has not occurred.
0b1	Underflow floating-point exception occurred during execution of the reported instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

OFF, bit [2]

Overflow floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

OFF	Meaning
0b0	Overflow floating-point exception has not occurred.
0b1	Overflow floating-point exception occurred during execution of the reported instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DZF, bit [1]

Divide by Zero floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

DZF	Meaning
0b0	Divide by Zero floating-point exception has not occurred.
0b1	Divide by Zero floating-point exception occurred during execution of the reported instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IOF, bit [0]

Invalid Operation floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

IOF	Meaning
0b0	Invalid Operation floating-point exception has not occurred.
0b1	Invalid Operation floating-point exception occurred during execution of the reported instruction.

The reset behavior of this field is:

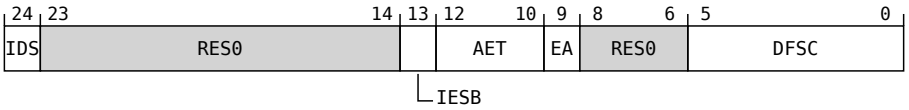
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from a trapped floating-point exception

In an implementation that supports the trapping of floating-point exceptions:

- From an Exception level using AArch64, the FPCR.{IDE, IXE, UFE, OFE, DZE, IOE} bits enable each of the floating-point exception traps.

ISS encoding for an SError exception



Note

In earlier versions of the architecture, an SError exception is referred to as an SError interrupt or an asynchronous External abort exception.

IDS, bit [24]

IMPLEMENTATION DEFINED syndrome.

IDS	Meaning
0b0	Bits [23:0] of the ISS field holds the fields described in this encoding. **Note** If FEAT_RAS is not implemented, bits [23:0] of the ISS field are RES0.
0b1	Bits [23:0] of the ISS field holds IMPLEMENTATION DEFINED syndrome information that can be used to provide additional information about the SError exception.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

This field was previously called ISV.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [23:14]

Reserved, RES0.

IESB, bit [13]

When FEAT_IESB is implemented and DFSC == 0b010001:

Implicit error synchronization event.

IESB	Meaning
0b0	The SError exception was either not synchronized by the implicit error synchronization event or not taken immediately.
0b1	The SError exception was synchronized by the implicit error synchronization event and taken immediately.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

AET, bits [12:10]

When FEAT_RAS is implemented and DFSC == 0b010001:

Asynchronous Error Type.

Describes the PE error state after taking the SError exception.

AET	Meaning
0b000	Uncontainable (UC).
0b001	Unrecoverable state (UEU).
0b010	Restartable state (UEO).
0b011	Recoverable state (UER).
0b110	Corrected (CE).

All other values are reserved.

If multiple errors are taken as a single SError exception, the overall PE error state is reported.

Note

Software can use this information to determine what recovery might be possible. The recovery software must also examine any implemented fault records to determine the location and extent of the error.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

EA, bit [9]

When FEAT_RAS is implemented and DFSC == 0b010001:

External abort type. Provides an IMPLEMENTATION DEFINED classification of External aborts.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [8:6]

Reserved, RES0.

DFSC, bits [5:0]

When FEAT_RAS is implemented:

Data Fault Status Code.

DFSC	Meaning
0b000000	Uncategorized error.
0b010001	Asynchronous SError exception.

All other values are reserved.

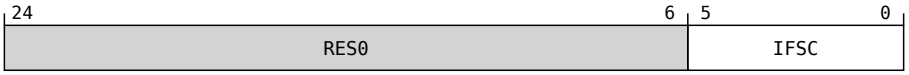
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ISS encoding for an exception from a Breakpoint or Vector Catch debug exception



Bits [24:6]

Reserved, RES0.

IFSC, bits [5:0]

Instruction Fault Status Code.

IFSC	Meaning
0b100010	Debug exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

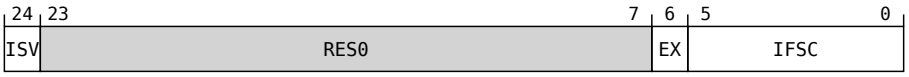
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Additional Information for ISS encoding for an exception from a Breakpoint or Vector Catch debug exception

For more information about generating these exceptions:

- For exceptions from AArch64, see 'Breakpoint exceptions'.

ISS encoding for an exception from a Software Step exception



ISV, bit [24]

Instruction syndrome valid. Indicates whether the EX bit, ISS[6], is valid, as follows:

ISV	Meaning
0b0	EX bit is RES0.
0b1	EX bit is valid.

See the EX bit description for more information.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [23:7]

Reserved, RES0.

EX, bit [6]

Exclusive operation. If the ISV bit is set to 1, this bit indicates whether a Load-Exclusive instruction was stepped.

EX	Meaning
0b0	An instruction other than a Load-Exclusive instruction was stepped.
0b1	A Load-Exclusive instruction was stepped.

If the ISV bit is set to 0, this bit is RES0, indicating no syndrome data is available.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IFSC, bits [5:0]

Instruction Fault Status Code.

IFSC	Meaning
0b100010	Debug exception.

The reset behavior of this field is:

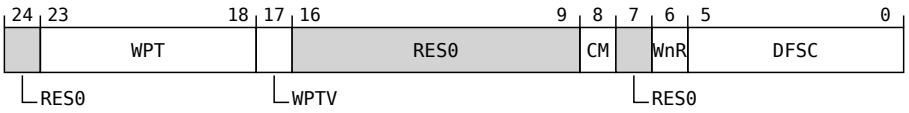
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from a Software Step exception

For more information about generating these exceptions, see 'Software Step exceptions'.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ISS encoding for an exception from a Watchpoint exception



Bit [24]

Reserved, RES0.

WPT, bits [23:18]

When FEAT_Debugv8p2 is implemented:

Watchpoint number.

All other values are reserved.

Otherwise:

Reserved, RES0.

WPTV, bit [17]

When FEAT_Debugv8p2 is implemented:

Watchpoint number Valid.

WPTV	Meaning
0b0	The WPT field is invalid, and holds an UNKNOWN value.
0b1	The WPT field is valid, and holds the number of a watchpoint that triggered a Watchpoint exception.

When a Watchpoint exception is triggered by a watchpoint match:

- If the PE sets any of FnV, FnP, or WPF to 1, then the PE sets WPTV to 1.
- Otherwise, the PE sets WPTV to an IMPLEMENTATION DEFINED value, 0 or 1.

Otherwise:

Reserved, RES0.

Bits [16:9]

Reserved, RES0.

CM, bit [8]

Cache maintenance. Indicates whether the Watchpoint exception came from a cache maintenance instruction:

CM	Meaning
0b0	The Watchpoint exception was not generated by the execution of one of the System instructions identified in the description of value 1.
0b1	The Watchpoint exception was generated by the execution of a cache maintenance instruction. The DC ZVA instruction is not classifed as a cache maintenance instruction, and therefore their execution does not cause this field to be set to 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bit [7]

Reserved, RES0.

WnR, bit [6]

Write not Read. Indicates whether the Watchpoint exception was caused by an instruction writing to a memory location, or by an instruction reading from a memory location.

WnR	Meaning
0b0	Watchpoint exception caused by an instruction reading from a memory location.
0b1	Watchpoint exception caused by an instruction writing to a memory location.

For Watchpoint exceptions on cache maintenance instructions, this bit always returns a value of 1.

For Watchpoint exceptions from an atomic instruction, this field is set to 0 if a read of the location would have generated the Watchpoint exception, otherwise it is set to 1.

If multiple watchpoints match on the same access, it is UNPREDICTABLE which watchpoint generates the Watchpoint exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

DFSC, bits [5:0]

Data Fault Status Code.

DFSC	Meaning
0b100010	Debug exception.

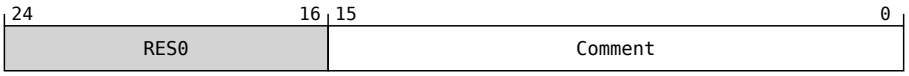
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from a Watchpoint exception

For more information about generating these exceptions, see 'Watchpoint exceptions'.

ISS encoding for an exception from execution of a Breakpoint instruction



Bits [24:16]

Reserved, RES0.

Comment, bits [15:0]

Set to the instruction comment field value, zero extended as necessary.

The reset behavior of this field is:

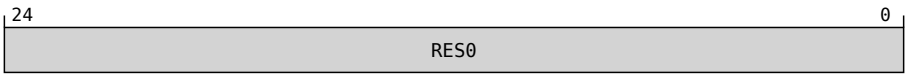
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from execution of a Breakpoint instruction

For more information about generating these exceptions, see 'Breakpoint instruction exceptions'.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ISS encoding for an exception from a trapped Pointer Authentication instruction



Bits [24:0]

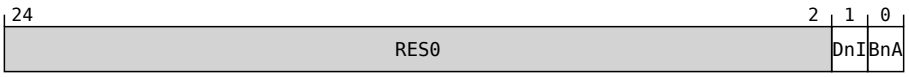
Reserved, RES0.

Additional Information for ISS encoding for an exception from a trapped Pointer Authentication instruction

For more information about generating these exceptions, see:

- HCR_EL2.API, for exceptions from Pointer authentication instructions, using AArch64 state, trapped to EL2.

ISS encoding for a PAC Fail exception



Bits [24:2]

Reserved, RES0.

DnI, bit [1]

This field indicates whether the exception is as a result of an Instruction key or a Data key.

DnI	Meaning
0b0	Instruction Key.
0b1	Data Key.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

BnA, bit [0]

This field indicates whether the exception is as a result of an A key or a B key.

BnA	Meaning
0b0	A key.
0b1	B key.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for a PAC Fail exception

The following instructions generate a PAC Fail exception when the Pointer Authentication Code (PAC) is incorrect:

- AUTDA, AUTDZA.
- AUTDB, AUTDZB.
- AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB.
- AUTIASPPC, AUTIASPPCR, AUTIA171615.
- AUTIBSPPC, AUTIBSPPCR, AUTIB171615.

If FEAT_FPACCOMBINE is implemented, the following instructions generate a PAC Fail exception when the Pointer Authentication Code (PAC) is incorrect:

- RETAA, RETAB.
- RETAASPPC, RETABSPPC.
- RETAASPPCR, RETABSPPCR.
- BLRAA, BLRAAZ, BLRAB, BLRABZ.
- BRAA, BRAB, BRAAZ, BRABZ.
- ERETAA, ERETAB.
- LDRAA, LDRAB, whether the authenticated address is written back to the base register or not.

Accessing ESR_EL1

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, ESR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ESR_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ESR_EL1;
```

MSR ESR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        ESR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    ESR_EL1 = X[t, 64];
```

MRS <Xt>, ESR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0101	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ESR_EL2;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR ESR_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0101	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    ESR_EL2 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.31 ESR_EL2, Exception Syndrome Register (EL2)

The ESR_EL2 characteristics are:

Purpose

Holds syndrome information for an exception taken to EL2.

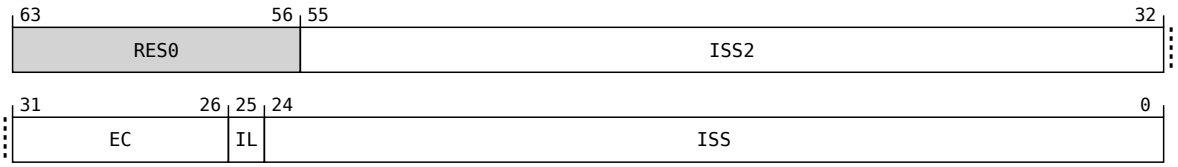
Configuration

There are no configuration notes.

Attributes

ESR_EL2 is a 64-bit register.

Field descriptions



ESR_EL2 is made UNKNOWN as a result of an exception return from EL2.

When an UNPREDICTABLE instruction is treated as UNDEFINED, and the exception is taken to EL2, the value of ESR_EL2 is UNKNOWN. The value written to ESR_EL2 must be consistent with a value that could be created as a result of an exception from the same Exception level that generated the exception as a result of a situation that is not UNPREDICTABLE at that Exception level, in order to avoid the possibility of a privilege violation.

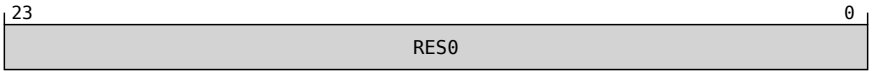
Bits [63:56]

Reserved, RES0.

ISS2, bits [55:32]

ISS2 encoding for an exception, the bit assignments are:

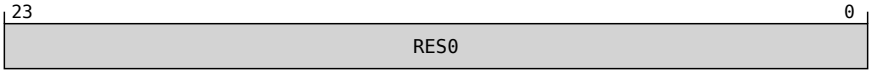
ISS2 encoding for an exception from a Data Abort



Bits [23:0]

Reserved, RES0.

ISS2 encoding for an exception from an Instruction Abort

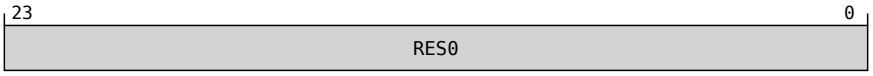


Bits [23:0]

Reserved, RES0.

ISS2 encoding for an exception from a Watchpoint exception

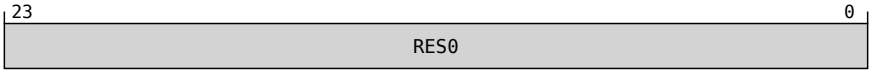
Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential



Bits [23:0]

Reserved, RES0.

ISS2 encoding for all other exceptions



Bits [23:0]

Reserved, RES0.

EC, bits [31:26]

Exception Class. Indicates the reason for the exception that this register holds information about.

For each EC value, the table references a subsection that gives information about:

- The cause of the exception, for example the configuration required to enable the trap.
- The encoding of the associated ISS.

Possible values of the EC field are:

EC	Meaning	ISS	ISS2	Applies when
0b000000	Unknown reason.	ISS encoding for exceptions with an unknown reason	ISS2 encoding for all other exceptions	
0b000001	Trapped WF* instruction execution. Conditional WF* instructions that fail their condition code check do not cause an exception.	ISS encoding for an exception from a WF* instruction	ISS2 encoding for all other exceptions	
0b000111	Access to Advanced SIMD or floating-point functionality trapped by CPACR_EL1.FPEN, CPTR_EL2.FPEN, CPTR_EL2.TFP, or CPTR_EL3.TFP control. Excludes exceptions resulting from CPACR_EL1 when the value of HCR_EL2.TGE is 1, or because Advanced SIMD and floating-point are not implemented. These are reported with EC value 0b000000.	ISS encoding for an exception from an access to a register or instruction resulting from the FPEN and TFP traps	ISS2 encoding for all other exceptions	
0b001001	Trapped use of a Pointer authentication instruction.	ISS encoding for an exception from a trapped Pointer Authentication instruction	ISS2 encoding for all other exceptions	FEAT_PAuth is implemented
0b001110	Illegal Execution state.	ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault	ISS2 encoding for all other exceptions	

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EC	Meaning	ISS	ISS2	Applies when
0b010101	SVC instruction execution in AArch64 state.	ISS encoding for an exception from HVC or SVC instruction execution	ISS2 encoding for all other exceptions	
0b010110	HVC instruction execution in AArch64 state, when HVC is not disabled.	ISS encoding for an exception from HVC or SVC instruction execution	ISS2 encoding for all other exceptions	
0b010111	SMC instruction execution in AArch64 state, when SMC is not disabled. This is reported in ESR_EL2 only when the exception is generated because the value of HCR_EL2.TSC is 1.	ISS encoding for an exception from SMC instruction execution in AArch64 state	ISS2 encoding for all other exceptions	
0b011000	Trapped MSR, MRS or System instruction execution in AArch64 state, that is not reported using EC values 0b000000, 0b000001 or 0b000111. This includes all instructions that cause exceptions that are part of the encoding space defined in 'System instruction class encoding overview', except for those exceptions reported using EC values 0b000000, 0b000001, or 0b000111.	ISS encoding for an exception from MSR, MRS, or System instruction execution in AArch64 state	ISS2 encoding for all other exceptions	
0b011100	Exception from a PAC Fail	ISS encoding for a PAC Fail exception	ISS2 encoding for all other exceptions	FEAT_FPAC is implemented
0b100000	Instruction Abort from a lower Exception level. Used for MMU and MPU faults generated by instruction accesses and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug-related exceptions.	ISS encoding for an exception from an Instruction Abort	ISS2 encoding for an exception from an Instruction Abort	
0b100001	Used for MMU and MPU faults generated by instruction accesses and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug-related exceptions.	ISS encoding for an exception from an Instruction Abort	ISS2 encoding for an exception from an Instruction Abort	
0b100010	PC alignment fault exception.	ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault	ISS2 encoding for all other exceptions	

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EC	Meaning	ISS	ISS2	Applies when
0b100100	Data Abort exception from a lower Exception level, excluding Data Abort exceptions taken to EL2 as a result of accesses generated associated with VNCR_EL2 as part of nested virtualization support. These Data Abort exceptions might be generated from Exception levels in any Execution state. Used for MMU and MPU faults generated by data accesses, alignment faults other than those caused by Stack Pointer misalignment, and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug-related exceptions.	ISS encoding for an exception from a Data Abort	ISS2 encoding for an exception from a Data Abort	
0b100101	Data Abort exception without a change in Exception level, or Data Abort exceptions taken to EL2 as a result of accesses generated associated with VNCR_EL2 as part of nested virtualization support. Used for MMU and MPU faults generated by data accesses, alignment faults other than those caused by Stack Pointer misalignment, and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug-related exceptions.	ISS encoding for an exception from a Data Abort	ISS2 encoding for an exception from a Data Abort	
0b100110	SP alignment fault exception.	ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault	ISS2 encoding for all other exceptions	
0b101100	Trapped floating-point exception taken from AArch64 state. This EC value is valid if the implementation supports trapping of floating-point exceptions, otherwise it is reserved. Whether a floating-point implementation supports trapping of floating-point exceptions is IMPLEMENTATION DEFINED.	ISS encoding for an exception from a trapped floating-point exception	ISS2 encoding for all other exceptions	
0b101111	SError exception.	ISS encoding for an SError exception	ISS2 encoding for all other exceptions	
0b110000	Breakpoint exception from a lower Exception level.	ISS encoding for an exception from a Breakpoint or Vector Catch debug exception	ISS2 encoding for all other exceptions	

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EC	Meaning	ISS	ISS2	Applies when
0b110001	Breakpoint exception taken without a change in Exception level.	ISS encoding for an exception from a Breakpoint or Vector Catch debug exception	ISS2 encoding for all other exceptions	
0b110010	Software Step exception from a lower Exception level.	ISS encoding for an exception from a Software Step exception	ISS2 encoding for all other exceptions	
0b110011	Software Step exception taken without a change in Exception level.	ISS encoding for an exception from a Software Step exception	ISS2 encoding for all other exceptions	
0b110100	Watchpoint from a lower Exception level, excluding Watchpoint Exceptions taken to EL2 as a result of accesses generated associated with VNCR_EL2 as part of nested virtualization support. These Watchpoint Exceptions might be generated from Exception levels using any Execution state.	ISS encoding for an exception from a Watchpoint exception	ISS2 encoding for an exception from a Watchpoint exception	
0b110101	Watchpoint exceptions without a change in Exception level, or Watchpoint exceptions taken to EL2 as a result of accesses generated associated with VNCR_EL2 as part of nested virtualization support.	ISS encoding for an exception from a Watchpoint exception	ISS2 encoding for an exception from a Watchpoint exception	
0b111100	BRK instruction execution in AArch64 state.	ISS encoding for an exception from execution of a Breakpoint instruction	ISS2 encoding for all other exceptions	

All other EC values are reserved by Arm, and:

- Unused values in the range 0b000000 - 0b101100 (0x00 - 0x2C) are reserved for future use for synchronous exceptions.
- Unused values in the range 0b101101 - 0b111111 (0x2D - 0x3F) are reserved for future use, and might be used for synchronous or asynchronous exceptions.

The effect of programming this field to a reserved value is that behavior is CONSTRAINED UNPREDICTABLE.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IL, bit [25]

Instruction Length for synchronous exceptions. Possible values of this bit are:

IL	Meaning
0b0	16-bit instruction trapped.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IL	Meaning

0b1	32-bit instruction trapped. This value is also used when the exception is one of the following:
	• An SError exception.
	• An Instruction Abort exception.
	• A PC alignment fault exception.
	• An SP alignment fault exception.
	• A Data Abort exception for which the value of the ISV bit is 0.
	• An Illegal Execution state exception.
	• Any debug exception except for Breakpoint instruction exceptions. For Breakpoint instruction exceptions, this bit has its standard meaning:
	  • 0b0: 16-bit T32 BKPT instruction.
	  • 0b1: 32-bit A32 BKPT instruction or A64 BRK instruction.
	• An exception reported using EC value 0b000000.

The reset behavior of this field is:

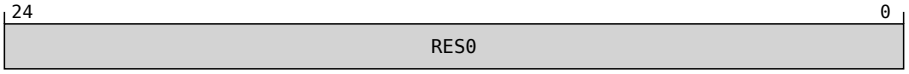
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ISS, bits [24:0]

Instruction Specific Syndrome. Architecturally, this field can be defined independently for each defined Exception class. However, in practice, some ISS encodings are used for more than one Exception class.

Typically, an ISS encoding has a number of subfields. When an ISS subfield holds a register number, the value returned in that field is the AArch64 view of the register number.

ISS encoding for exceptions with an unknown reason



Bits [24:0]

Reserved, RES0.

Additional Information for ISS encoding for exceptions with an unknown reason

When an exception is reported using this EC value, the IL field is set to 1.

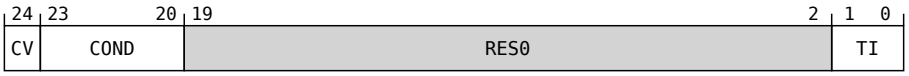
This EC value is used for all exceptions that are not covered by any other EC value. This includes exceptions that are generated in the following situations:

- The attempted execution of an instruction bit pattern that has no allocated instruction or that is not accessible at the current Exception level and Security state, including:
  - A read access using a System register pattern that is not allocated for reads or that does not permit reads at the current Exception level and Security state.
  - A write access using a System register pattern that is not allocated for writes or that does not permit writes at the current Exception level and Security state.
  - Instruction encodings that are unallocated.
  - Instruction encodings for instructions or System registers that are not implemented in the implementation.
- In Debug state, the attempted execution of an instruction bit pattern that is not accessible in Debug state.
- In Non-debug state, the attempted execution of an instruction bit pattern that is not accessible in Non-debug state.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- In an implementation that does not include Advanced SIMD and floating-point functionality, an attempted access to Advanced SIMD or floating-point functionality under conditions where that access would be permitted if that functionality was present. This includes the attempted execution of an Advanced SIMD or floating-point instruction, and attempted accesses to Advanced SIMD and floating-point System registers.
- An exception generated because of the value of one of the SCTLR_EL1.{ITD, SED, CP15BEN} control bits.
- Attempted execution of:
  - An HVC instruction when disabled by HCR_EL2.HCD.
  - An SMC instruction when disabled by HCR_EL2.TSC.
  - An HLT instruction when disabled by EDSCR.HDE.
- Attempted execution of an MSR or MRS instruction to access SP_EL0 when the value of SPSel.SP is 0.
- Attempted execution of an MSR or MRS instruction using a _EL12 register name.
- Attempted execution, in Debug state, of:
  - A DCPS1 instruction when the value of HCR_EL2.TGE is 1.
  - A DCPS2 instruction from EL1 or EL0 when EL2 is disabled or not implemented in the current Security state.
  - A DCPS3 instruction when the value of EDSCR.SDD is 1.
- In Debug state when the value of EDSCR.SDD is 1, the attempted execution at EL1 or EL0 of an instruction that is configured to trap to EL2.
- An exception that is taken to EL2 because the value of HCR_EL2.TGE is 1. If the value of HCR_EL2.TGE is 0, this exception is reported using an ESR_EL2.EC value of `0b000111`.

ISS encoding for an exception from a WF* instruction



CV, bit [24]

Condition code valid.

CV	Meaning
`0b0`	The COND field is not valid.
`0b1`	The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

For exceptions taken from AArch64, this field is set to `0b1110`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [19:2]

Reserved, RES0.

TI, bits [1:0]

Trapped instruction. Possible values of this bit are:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TI	Meaning
0b00	WFI trapped.
0b01	WFE trapped.

The reset behavior of this field is:

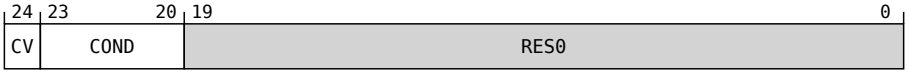
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from a WF* instruction

The following fields describe configuration settings for generating this exception:

- SCTLR_EL1.{nTWE, nTWI}.
- SCTLR_EL2.{nTWE, nTWI}.
- HCR_EL2.{TWE, TWI}.

ISS encoding for an exception from an access to a register or instruction resulting from the FPEN and TFP traps



The accesses covered by this trap include:

- Execution of Advanced SIMD and floating-point instructions.
- Accesses to the Advanced SIMD and floating-point System registers.

For an implementation that does not include support for Advanced SIMD and floating-point, the exception is reported using the EC value 0b000000.

CV, bit [24]

Condition code valid.

CV	Meaning
0b0	The COND field is not valid.
0b1	The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

For exceptions taken from AArch64, this field is set to 0b1110.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [19:0]

Reserved, RES0.

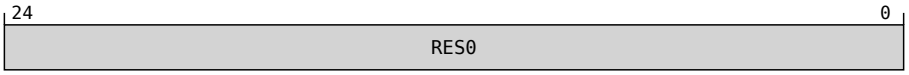
Additional Information for ISS encoding for an exception from an access to a register or instruction resulting from the FPEN and TFP traps

The following fields describe the configuration settings for the traps that are reported using EC value 0b000111:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- CPACR_EL1.FPEN, for accesses to SIMD and floating-point registers trapped to EL1.
- CPTR_EL2.FPEN and CPTR_EL2.TFP, for accesses to SIMD and floating-point registers trapped to EL2.

ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault



Bits [24:0]

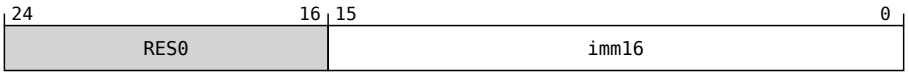
Reserved, RES0.

Additional Information for ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault

There are no configuration settings for generating Illegal Execution state exceptions and PC alignment fault exceptions. For more information about PC alignment fault exceptions, see 'PC alignment checking'.

'SP alignment checking' describes the configuration settings for generating SP alignment fault exceptions.

ISS encoding for an exception from HVC or SVC instruction execution



Bits [24:16]

Reserved, RES0.

imm16, bits [15:0]

The value of the immediate field from the HVC or SVC instruction.

For an HVC instruction, and for an A64 SVC instruction, this is the value of the imm16 field of the issued instruction.

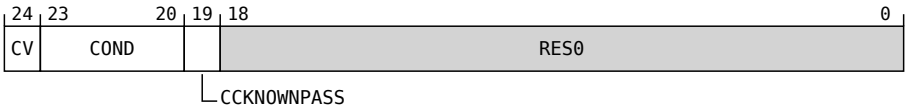
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from HVC or SVC instruction execution

For A64 instructions, see 'SVC' and 'HVC'.

ISS encoding for an exception from SMC instruction execution in AArch32 state



For an SMC instruction that completes normally and generates an exception that is taken to EL3, the ISS encoding is RES0.

For an SMC instruction that is trapped to EL2 from EL1 because HCR_EL2.TSC is 1, the ISS encoding is as shown in the diagram.

CV, bit [24]

Condition code valid.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CV	Meaning
0b0	The COND field is not valid.
0b1	The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

This field is valid only if CCKNOWNPASS is 1, otherwise it is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

For exceptions taken from AArch64, this field is set to 0b1110.

This field is valid only if CCKNOWNPASS is 1, otherwise it is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CCKNOWNPASS, bit [19]

Indicates whether the instruction might have failed its condition code check.

CCKNOWNPASS	Meaning
0b0	The instruction was unconditional, or was conditional and passed its condition code check.
0b1	The instruction was conditional, and might have failed its condition code check.

Note

In an implementation in which an SMC instruction that fails it code check is not trapped, this field can always return the value 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

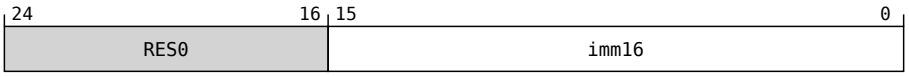
Bits [18:0]

Reserved, RES0.

Additional Information for ISS encoding for an exception from SMC instruction execution in AArch32 state

HCR_EL2.TSC describes the configuration settings for trapping SMC instructions to EL2.

ISS encoding for an exception from SMC instruction execution in AArch64 state



Bits [24:16]

Reserved, RES0.

imm16, bits [15:0]

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The value of the immediate field from the issued SMC instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

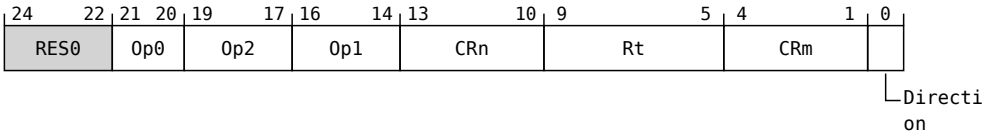
Additional Information for ISS encoding for an exception from SMC instruction execution in AArch64 state

The value of ISS[24:0] described here is used both:

- When an SMC instruction is trapped from EL1 modes.
- When an SMC instruction is not trapped, so completes normally and generates an exception that is taken to EL3.

HCR_EL2.TSC describes the configuration settings for trapping SMC from EL1 modes.

ISS encoding for an exception from MSR, MRS, or System instruction execution in AArch64 state



Bits [24:22]

Reserved, RES0.

Op0, bits [21:20]

The Op0 value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Op2, bits [19:17]

The Op2 value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Op1, bits [16:14]

The Op1 value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CRn, bits [13:10]

The CRn value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Rt, bits [9:5]

The Rt value from the issued instruction, the general-purpose register used for the transfer.

For system instructions which require that the opcode Rt field is set to `0b11111`, but where the trapped instruction has a different value of Rt, an implementation is permitted to return the value `0b11111`, instead of the value of Rt from the trapped instruction.

The reset behavior of this field is:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CRm, bits [4:1]

The CRm value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Direction, bit [0]

Indicates the direction of the trapped instruction.

Direction	Meaning
0b0	Write access, including MSR instructions.
0b1	Read access, including MRS instructions.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from MSR, MRS, or System instruction execution in AArch64 state

For exceptions caused by System instructions, see 'System instructions' subsection of 'Branches, exception generating and System instructions' for the encoding values returned by an instruction.

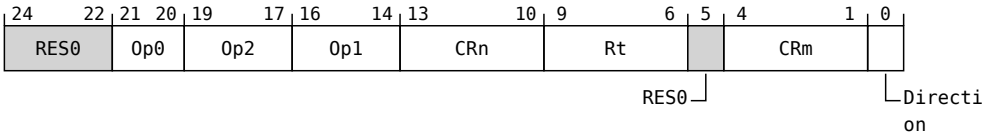
The following fields describe configuration settings for generating the exception that is reported using EC value 0b011000:

- SCTLR_EL1.UCI, for execution of cache maintenance instructions using AArch64 state, execution is trapped to EL1 or EL2.
- SCTLR_EL1.UCT, for accesses to CTR_EL0 using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- SCTLR_EL1.DZE, for execution of DC ZVA instructions using AArch64 state, execution is trapped to EL1 or EL2.
- SCTLR_EL1.UMA, for accesses to the PSTATE interrupt masks using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- CPACR_EL1.TTA, for accesses to the trace registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- MDSCR_EL1.TDCC, for accesses to the Debug Communications Channel (DCC) registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- CNTKCTL_EL1.{EL0PTEN, EL0VTEN, EL0PCTEN, EL0VCTEN} accesses to the Generic Timer registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- PMUSERENR_EL0, for accesses to the Performance Monitor registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- AMUSERENR_EL0.EN, for accesses to Activity Monitors registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- HCR_EL2.{TRVM, TVM}, for accesses to virtual memory control registers using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.TDZ, for execution of DC ZVA instructions using AArch64 state, execution is trapped to EL2.
- HCR_EL2.TTLB, for execution of TLB maintenance instructions using AArch64 state, execution is trapped to EL2.
- HCR_EL2.{TSW, TPC, TPU}, for execution of cache maintenance instructions using AArch64 state, execution is trapped to EL2.
- HCR_EL2.TACR, for accesses to the Auxiliary Control Register, ACTLR_EL1, using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.TIDCP, for accesses to lockdown, DMA, and TCM operations using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.{TID1, TID2, TID3}, for accesses to ID group 1, ID group 2 or ID group 3 registers, using AArch64 state, MSR or MRS access trapped to EL2.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- CPTR_EL2.TCPAC, for accesses to CPACR_EL1, using AArch64 state, MSR or MRS access trapped to EL2.
- CPTR_EL2.TTA, for accesses to the trace registers, using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.TTRF, for accesses to the trace filter control register, TRFCR_EL1, using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.TDRA, for accesses to Debug ROM registers, using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.TDOSA, for accesses to powerdown debug registers using AArch64 state, MSR or MRS access trapped to EL2.
- CNTHCTL_EL2.{EL1PCEN, EL1PCTEN}, for accesses to the Generic Timer registers using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.TDA, for accesses to debug registers using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.{TPM, TPMCR}, for accesses to Performance Monitor registers, using AArch64 state, MSR or MRS access trapped to EL2.
- CPTR_EL2.TAM, for accesses to Activity Monitors registers, using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.APK, for accesses to Pointer authentication key registers, using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.{NV, NV1}, for Nested virtualization register access, using AArch64 state, MSR or MRS access, trapped to EL2.
- HCR_EL2.AT, for execution of AT S1E* instructions, using AArch64 state, execution is trapped to EL2.
- HCR_EL2.{TERR, FIEN}, for accesses to RAS registers, using AArch64 state, MSR or MRS access, trapped to EL2.
- CPTR_EL3.TAM, for accesses to Activity Monitors registers, using AArch64 state, MSR or MRS access, trapped to EL3.

ISS encoding for an exception from MSRR, MRRS, or 128-bit System instruction execution in AArch64 state



Bits [24:22]

Reserved, RES0.

Op0, bits [21:20]

The Op0 value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Op2, bits [19:17]

The Op2 value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Op1, bits [16:14]

The Op1 value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CRn, bits [13:10]

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The CRn value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Rt, bits [9:6]

The Rt value from the issued instruction, the general-purpose register used for the transfer.

Note

This value represents register pair of X[Rt:0], X[Rt:1].

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [5]

Reserved, RES0.

CRm, bits [4:1]

The CRm value from the issued instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Direction, bit [0]

Indicates the direction of the trapped instruction.

Direction	Meaning
0b0	Write access, MSRR instructions.
0b1	Read access, MRRS instructions.

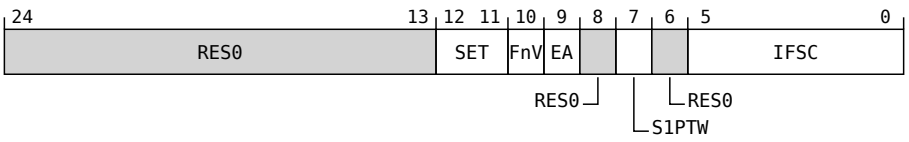
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from MSRR, MRRS, or 128-bit System instruction execution in AArch64 state

The following fields describe configuration settings for generating exceptions from an MSRR or MRRS access that are reported using EC value 0b010100:

ISS encoding for an exception from an Instruction Abort



Bits [24:13]

Reserved, RES0.

SET, bits [12:11]

When FEAT_RAS is implemented and IFSC == 0b010000:

Synchronous Error Type. When IFSC is 0b010000, describes the PE error state after taking the Instruction Abort exception.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SET	Meaning
0b00	Recoverable state (UER).
0b10	Uncontainable (UC).
0b11	Restartable state (UEO).

All other values are reserved.

Note

Software can use this information to determine what recovery might be possible. Taking a synchronous External abort exception might result in a PE state that is not recoverable.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

FnV, bit [10]

When IFSC == 0b010000:

FAR not Valid, for a synchronous External abort other than a synchronous External abort on a translation table walk.

FnV	Meaning
0b0	FAR is valid.
0b1	FAR is not valid, and holds an UNKNOWN value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EA, bit [9]

External abort type. This bit can provide an IMPLEMENTATION DEFINED classification of External aborts.

For any abort other than an External abort this bit returns a value of 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [8]

Reserved, RES0.

S1PTW, bit [7]

For a stage 2 fault, indicates whether the fault was a stage 2 fault on an access made for a stage 1 translation table walk:

S1PTW	Meaning
0b0	Fault not on a stage 2 translation for a stage 1 translation table walk.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

S1PTW	Meaning
0b1	Fault on the stage 2 translation of an access for a stage 1 translation table walk.

For any abort other than a stage 2 fault this bit is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [6]

Reserved, RES0.

IFSC, bits [5:0]

Instruction Fault Status Code.

IFSC	Meaning	Applies when
0b000000	Address size fault, level 0 of translation or translation table base register.	
0b000001	Address size fault, level 1.	
0b000010	Address size fault, level 2.	
0b000011	Address size fault, level 3.	
0b000100	Translation fault, level 0.	
0b000101	Translation fault, level 1.	
0b000110	Translation fault, level 2.	
0b000111	Translation fault, level 3.	
0b001001	Access flag fault, level 1.	
0b001010	Access flag fault, level 2.	
0b001011	Access flag fault, level 3.	
0b001100	Permission fault, level 0.	
0b001101	Permission fault, level 1.	
0b001110	Permission fault, level 2.	
0b001111	Permission fault, level 3.	
0b010000	Synchronous External abort, not on translation table walk or hardware update of translation table.	
0b010100	Synchronous External abort on translation table walk or hardware update of translation table, level 0.	
0b010101	Synchronous External abort on translation table walk or hardware update of translation table, level 1.	
0b010110	Synchronous External abort on translation table walk or hardware update of translation table, level 2.	
0b010111	Synchronous External abort on translation table walk or hardware update of translation table, level 3.	

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IFSC	Meaning	Applies when
0b011000	Synchronous parity or ECC error on memory access, not on translation table walk.	FEAT_RAS is not implemented
0b011100	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 0.	FEAT_RAS is not implemented
0b011101	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 1.	FEAT_RAS is not implemented
0b011110	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 2.	FEAT_RAS is not implemented
0b011111	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 3.	FEAT_RAS is not implemented
0b110000	TLB conflict abort.	
0b110001	Unsupported atomic hardware update fault.	FEAT_HAFDBS is implemented

All other values are reserved.

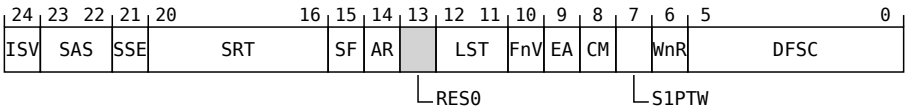
For more information about the lookup level associated with a fault, see 'The lookup level associated with MMU faults'.

If the S1PTW bit is set, then the level refers the level of the stage2 translation that is translating a stage 1 translation walk.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ISS encoding for an exception from a Data Abort



ISV, bit [24]

Instruction Syndrome Valid. Indicates whether the syndrome information in ISS[23:14] is valid.

ISV	Meaning
0b0	No valid instruction syndrome. ISS[23:14] are RES0.
0b1	ISS[23:14] hold a valid instruction syndrome.

This bit is 0 for all faults reported in ESR_EL2 except the following stage 2 aborts:

- AArch64 loads and stores of a single general-purpose register (including the register specified with 0b11111, including those with Acquire/Release semantics, but excluding Load Exclusive or Store Exclusive and excluding those with writeback).

For these stage 2 aborts, ISV is UNKNOWN if the exception was generated in Debug state in Memory access mode, and otherwise indicates whether ISS[23:14] hold a valid syndrome.

ISV is 0 for all faults reported in ESR_EL1 or ESR_EL3.

When FEAT_RAS is implemented, ISV is 0 for any synchronous External abort.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_RAS is not implemented, it is IMPLEMENTATION DEFINED whether ISV is set to 1 or 0 on a synchronous External abort on a stage 2 translation table walk.

For ISS reporting, a stage 2 abort on a stage 1 translation table walk does not return a valid instruction syndrome, and therefore ISV is 0 for these aborts.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SAS, bits [23:22]

When ISV == '1':

Syndrome Access Size. Indicates the size of the access attempted by the faulting operation.

SAS	Meaning
0b00	Byte
0b01	Halfword
0b10	Word
0b11	Doubleword

This field is UNKNOWN when the value of ISV is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SSE, bit [21]

When ISV == '1':

Syndrome Sign Extend. For a byte, halfword, or word load operation, indicates whether the data item must be sign extended.

SSE	Meaning
0b0	Sign-extension not required.
0b1	Data item must be sign-extended.

For all other operations, this field is 0.

This field is UNKNOWN when the value of ISV is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SRT, bits [20:16]

When ISV == '1':

Syndrome Register Transfer. The register number of the Wt/Xt/Rt operand of the faulting instruction.

This field is UNKNOWN when the value of ISV is UNKNOWN.

The reset behavior of this field is:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [15]

When ISV == '1':

SF, bit [15]

Sixty Four bit general-purpose register transfer. Width of the register accessed by the instruction is 64-bit.

SF	Meaning
0b0	Instruction loads/stores a 32-bit general-purpose register.
0b1	Instruction loads/stores a 64-bit general-purpose register.

Note

This field specifies the register width identified by the instruction, not the Execution state.

This field is UNKNOWN when the value of ISV is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

FnP, bit [15]

FAR not Precise.

FnP	Meaning
0b0	The FAR holds the faulting virtual address that generated the Data Abort.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

AR, bit [14]

When ISV == '1':

Acquire/Release.

AR	Meaning
0b0	Instruction did not have acquire/release semantics.
0b1	Instruction did have acquire/release semantics.

This field is UNKNOWN when the value of ISV is UNKNOWN.

For a load-acquire instruction that does not have acquire semantics as the result of the destination register being ZR, it is IMPLEMENTATION SPECIFIC whether this field is reported as 0 or 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bit [13]

Reserved, RES0.

Bits [12:11]

When (DFSC IN {0b00xxxx} || DFSC IN {0b10101x}) && !(DFSC IN {0b0000xx}):

LST, bits [12:11]

Load/Store Type. Used when a Translation fault, Access flag fault, or Permission fault generates a Data Abort.

LST	Meaning
0b00	The instruction that generated the Data Abort is not specified by this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When IsFeatureImplemented(FEAT_RAS) && ((DFSC == 0b010000 || DFSC IN {0b01001x}) || DFSC IN {0b0101xx}):

SET, bits [12:11]

Synchronous Error Type. Used when a synchronous External abort, not on a Translation table walk or hardware update of the Translation table, generated the Data Abort. Describes the PE error state after taking the Data Abort exception.

SET	Meaning
0b00	Recoverable state (UER).
0b11	Restartable state (UEO).

Note

Software can use this information to determine what recovery might be possible. Taking a synchronous External abort exception might result in a PE state that is not recoverable.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

FnV, bit [10]

FAR not Valid, for a synchronous External abort other than a synchronous External abort on a translation table walk.

FnV	Meaning
0b0	FAR is valid.
0b1	FAR is not valid, and holds an UNKNOWN value.

This field is valid only if the DFSC code is 0b010000. It is RES0 for all other aborts.

The reset behavior of this field is:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EA, bit [9]

External abort type. This bit can provide an IMPLEMENTATION DEFINED classification of External aborts.

For any abort other than an External abort this bit returns a value of 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CM, bit [8]

Cache maintenance. Indicates whether the Data Abort came from a cache maintenance or address translation instruction:

CM	Meaning
0b0	The Data Abort was not generated by the execution of one of the System instructions identified in the description of value 1.
0b1	The Data Abort was generated by either the execution of a cache maintenance instruction or by a synchronous fault on the execution of an address translation instruction. The DC ZVA, DC GVA, and DC GZVA instructions are not classified as cache maintenance instructions, and therefore their execution cannot cause this field to be set to 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

S1PTW, bit [7]

For a stage 2 fault, indicates whether the fault was a stage 2 fault on an access made for a stage 1 translation table walk:

S1PTW	Meaning
0b0	Fault not on a stage 2 translation for a stage 1 translation table walk.
0b1	Fault on the stage 2 translation of an access for a stage 1 translation table walk.

For any abort other than a stage 2 fault this bit is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

WnR, bit [6]

Write not Read. Indicates whether a synchronous abort was caused by an instruction writing to a memory location, or by an instruction reading from a memory location.

WnR	Meaning
0b0	Abort caused by an instruction reading from a memory location.
0b1	Abort caused by an instruction writing to a memory location.

For faults on cache maintenance and address translation instructions, this bit always returns a value of 1.

For faults from an atomic instruction that both reads and writes from a memory location, this bit is set to 0 if a read of the address specified by the instruction would have generated the fault which is being reported, otherwise it is set to 1. The architecture permits, but does not require, a relaxation of this requirement such that for all stage 2 aborts on stage 1 translation table walks for atomic instructions, the WnR bit is always 0.

This field is UNKNOWN for:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- An External abort on an Atomic access.
- A fault reported using a DFSC value of `0b110101` or `0b110001`, indicating an unsupported Exclusive or atomic access.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

DFSC, bits [5:0]

Data Fault Status Code.

DFSC	Meaning	Applies when
`0b000000`	Address size fault, level 0 of translation or translation table base register.	
`0b000001`	Address size fault, level 1.	
`0b000010`	Address size fault, level 2.	
`0b000011`	Address size fault, level 3.	
`0b000100`	Translation fault, level 0.	
`0b000101`	Translation fault, level 1.	
`0b000110`	Translation fault, level 2.	
`0b000111`	Translation fault, level 3.	
`0b001001`	Access flag fault, level 1.	
`0b001010`	Access flag fault, level 2.	
`0b001011`	Access flag fault, level 3.	
`0b001100`	Permission fault, level 0.	
`0b001101`	Permission fault, level 1.	
`0b001110`	Permission fault, level 2.	
`0b001111`	Permission fault, level 3.	
`0b010000`	Synchronous External abort, not on translation table walk or hardware update of translation table.	
`0b010100`	Synchronous External abort on translation table walk or hardware update of translation table, level 0.	
`0b010101`	Synchronous External abort on translation table walk or hardware update of translation table, level 1.	
`0b010110`	Synchronous External abort on translation table walk or hardware update of translation table, level 2.	
`0b010111`	Synchronous External abort on translation table walk or hardware update of translation table, level 3.	
`0b011000`	Synchronous parity or ECC error on memory access, not on translation table walk.	FEAT_RAS is not implemented
`0b011100`	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 0.	FEAT_RAS is not implemented
`0b011101`	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 1.	FEAT_RAS is not implemented
`0b011110`	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 2.	FEAT_RAS is not implemented

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DFSC	Meaning	Applies when
0b011111	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 3.	FEAT_RAS is not implemented
0b100001	Alignment fault.	
0b110000	TLB conflict abort.	
0b110001	Unsupported atomic hardware update fault.	FEAT_HAFDBS is implemented
0b110100	IMPLEMENTATION DEFINED fault (Lockdown).	
0b110101	IMPLEMENTATION DEFINED fault (Unsupported Exclusive or Atomic access).	

All other values are reserved.

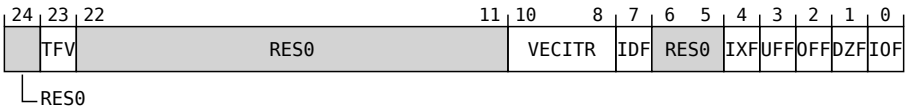
For more information about the lookup level associated with a fault, see 'The lookup level associated with MMU faults'.

If the S1PTW bit is set, then the level refers the level of the stage2 translation that is translating a stage 1 translation walk.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ISS encoding for an exception from a trapped floating-point exception



Bit [24]

Reserved, RES0.

TFV, bit [23]

Trapped Fault Valid bit. Indicates whether the IDF, IXF, UFF, OFF, DZF, and IOF bits hold valid information about trapped floating-point exceptions.

TFV	Meaning
0b0	The IDF, IXF, UFF, OFF, DZF, and IOF bits do not hold valid information about trapped floating-point exceptions and are UNKNOWN.
0b1	One or more floating-point exceptions occurred during an operation performed while executing the reported instruction. The IDF, IXF, UFF, OFF, DZF, and IOF bits indicate trapped floating-point exceptions that occurred. For more information, see 'Floating-point exceptions and exception traps'.

It is IMPLEMENTATION DEFINED whether this field is set to 0 on an exception generated by a trapped floating-point exception from an instruction that is performing floating-point operations on more than one lane of a vector.

Note

This is not a requirement. Implementations can set this field to 1 on a trapped floating-point exception from an instruction and return valid information in the {IDF, IXF, UFF, OFF, DZF, IOF} fields.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [22:11]

Reserved, RES0.

VECITR, bits [10:8]

For a trapped floating-point exception from an instruction executed in AArch64 state this field is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IDF, bit [7]

Input Denormal floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

IDF	Meaning
0b0	Input denormal floating-point exception has not occurred.
0b1	Input denormal floating-point exception occurred during execution of the reported instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [6:5]

Reserved, RES0.

IXF, bit [4]

Inexact floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

IXF	Meaning
0b0	Inexact floating-point exception has not occurred.
0b1	Inexact floating-point exception occurred during execution of the reported instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

UFF, bit [3]

Underflow floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

UFF	Meaning
0b0	Underflow floating-point exception has not occurred.
0b1	Underflow floating-point exception occurred during execution of the reported instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

OFF, bit [2]

Overflow floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

OFF	Meaning
0b0	Overflow floating-point exception has not occurred.
0b1	Overflow floating-point exception occurred during execution of the reported instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

DZF, bit [1]

Divide by Zero floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

DZF	Meaning
0b0	Divide by Zero floating-point exception has not occurred.
0b1	Divide by Zero floating-point exception occurred during execution of the reported instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IOF, bit [0]

Invalid Operation floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

IOF	Meaning
0b0	Invalid Operation floating-point exception has not occurred.
0b1	Invalid Operation floating-point exception occurred during execution of the reported instruction.

The reset behavior of this field is:

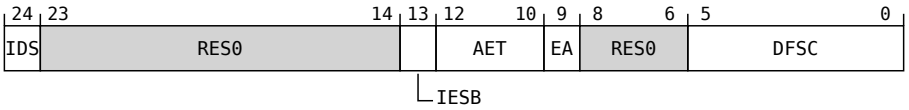
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from a trapped floating-point exception

In an implementation that supports the trapping of floating-point exceptions:

- From an Exception level using AArch64, the FPCR.{IDE, IXE, UFE, OFE, DZE, IOE} bits enable each of the floating-point exception traps.

ISS encoding for an SError exception



Note

In earlier versions of the architecture, an SError exception is referred to as an SError interrupt or an asynchronous External abort exception.

IDS, bit [24]

IMPLEMENTATION DEFINED syndrome.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IDS	Meaning
0b0	Bits [23:0] of the ISS field holds the fields described in this encoding. **Note** If FEAT_RAS is not implemented, bits [23:0] of the ISS field are RES0.
0b1	Bits [23:0] of the ISS field holds IMPLEMENTATION DEFINED syndrome information that can be used to provide additional information about the SError exception.

Note

This field was previously called ISV.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [23:14]

Reserved, RES0.

IESB, bit [13]

When FEAT_IESB is implemented and DFSC == 0b010001:

Implicit error synchronization event.

IESB	Meaning
0b0	The SError exception was either not synchronized by the implicit error synchronization event or not taken immediately.
0b1	The SError exception was synchronized by the implicit error synchronization event and taken immediately.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

AET, bits [12:10]

When FEAT_RAS is implemented and DFSC == 0b010001:

Asynchronous Error Type.

Describes the PE error state after taking the SError exception.

AET	Meaning
0b000	Uncontainable (UC).
0b001	Unrecoverable state (UEU).
0b010	Restartable state (UEO).
0b011	Recoverable state (UER).
0b110	Corrected (CE).

All other values are reserved.

If multiple errors are taken as a single SError exception, the overall PE error state is reported.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Software can use this information to determine what recovery might be possible. The recovery software must also examine any implemented fault records to determine the location and extent of the error.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EA, bit [9]

When FEAT_RAS is implemented and DFSC == 0b010001:

External abort type. Provides an IMPLEMENTATION DEFINED classification of External aborts.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [8:6]

Reserved, RES0.

DFSC, bits [5:0]

When FEAT_RAS is implemented:

Data Fault Status Code.

DFSC	Meaning
0b000000	Uncategorized error.
0b010001	Asynchronous SError exception.

All other values are reserved.

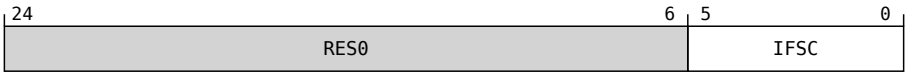
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ISS encoding for an exception from a Breakpoint or Vector Catch debug exception



Bits [24:6]

Reserved, RES0.

IFSC, bits [5:0]

Instruction Fault Status Code.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IFSC	Meaning
0b100010	Debug exception.

The reset behavior of this field is:

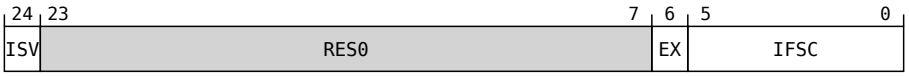
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from a Breakpoint or Vector Catch debug exception

For more information about generating these exceptions:

- For exceptions from AArch64, see 'Breakpoint exceptions'.

ISS encoding for an exception from a Software Step exception



ISV, bit [24]

Instruction syndrome valid. Indicates whether the EX bit, ISS[6], is valid, as follows:

ISV	Meaning
0b0	EX bit is RES0.
0b1	EX bit is valid.

See the EX bit description for more information.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [23:7]

Reserved, RES0.

EX, bit [6]

Exclusive operation. If the ISV bit is set to 1, this bit indicates whether a Load-Exclusive instruction was stepped.

EX	Meaning
0b0	An instruction other than a Load-Exclusive instruction was stepped.
0b1	A Load-Exclusive instruction was stepped.

If the ISV bit is set to 0, this bit is RES0, indicating no syndrome data is available.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IFSC, bits [5:0]

Instruction Fault Status Code.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IFSC	Meaning
0b100010	Debug exception.

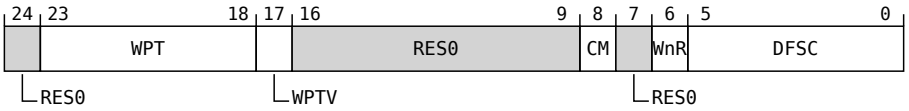
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from a Software Step exception

For more information about generating these exceptions, see 'Software Step exceptions'.

ISS encoding for an exception from a Watchpoint exception



Bit [24]

Reserved, RES0.

WPT, bits [23:18]

When FEAT_Debugv8p2 is implemented:

Watchpoint number.

All other values are reserved.

Otherwise:

Reserved, RES0.

WPTV, bit [17]

When FEAT_Debugv8p2 is implemented:

Watchpoint number Valid.

WPTV	Meaning
0b0	The WPT field is invalid, and holds an UNKNOWN value.
0b1	The WPT field is valid, and holds the number of a watchpoint that triggered a Watchpoint exception.

When a Watchpoint exception is triggered by a watchpoint match:

- If the PE sets any of FnV, FnP, or WPF to 1, then the PE sets WPTV to 1.
- Otherwise, the PE sets WPTV to an IMPLEMENTATION DEFINED value, 0 or 1.

Otherwise:

Reserved, RES0.

Bits [16:9]

Reserved, RES0.

CM, bit [8]

Cache maintenance. Indicates whether the Watchpoint exception came from a cache maintenance instruction:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CM	Meaning
0b0	The Watchpoint exception was not generated by the execution of one of the System instructions identified in the description of value 1.
0b1	The Watchpoint exception was generated by the execution of a cache maintenance instruction. The DC ZVA instruction is not classifed as a cache maintenance instruction, and therefore their execution does not cause this field to be set to 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [7]

Reserved, RES0.

WnR, bit [6]

Write not Read. Indicates whether the Watchpoint exception was caused by an instruction writing to a memory location, or by an instruction reading from a memory location.

WnR	Meaning
0b0	Watchpoint exception caused by an instruction reading from a memory location.
0b1	Watchpoint exception caused by an instruction writing to a memory location.

For Watchpoint exceptions on cache maintenance instructions, this bit always returns a value of 1.

For Watchpoint exceptions from an atomic instruction, this field is set to 0 if a read of the location would have generated the Watchpoint exception, otherwise it is set to 1.

If multiple watchpoints match on the same access, it is UNPREDICTABLE which watchpoint generates the Watchpoint exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

DFSC, bits [5:0]

Data Fault Status Code.

DFSC	Meaning
0b100010	Debug exception.

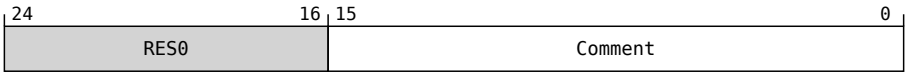
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from a Watchpoint exception

For more information about generating these exceptions, see 'Watchpoint exceptions'.

ISS encoding for an exception from execution of a Breakpoint instruction



Bits [24:16]

Reserved, RES0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Comment, bits [15:0]

Set to the instruction comment field value, zero extended as necessary.

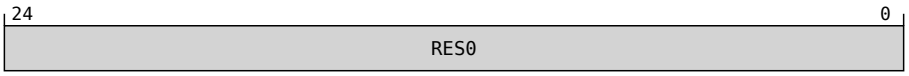
The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for an exception from execution of a Breakpoint instruction

For more information about generating these exceptions, see 'Breakpoint instruction exceptions'.

ISS encoding for an exception from a trapped Pointer Authentication instruction



Bits [24:0]

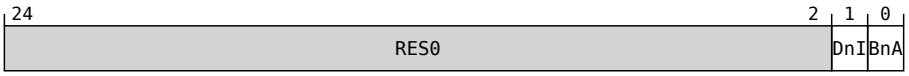
Reserved, RES0.

Additional Information for ISS encoding for an exception from a trapped Pointer Authentication instruction

For more information about generating these exceptions, see:

- HCR_EL2.API, for exceptions from Pointer authentication instructions, using AArch64 state, trapped to EL2.

ISS encoding for a PAC Fail exception



Bits [24:2]

Reserved, RES0.

DnI, bit [1]

This field indicates whether the exception is as a result of an Instruction key or a Data key.

DnI	Meaning
0b0	Instruction Key.
0b1	Data Key.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

BnA, bit [0]

This field indicates whether the exception is as a result of an A key or a B key.

BnA	Meaning
0b0	A key.
0b1	B key.

The reset behavior of this field is:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Additional Information for ISS encoding for a PAC Fail exception

The following instructions generate a PAC Fail exception when the Pointer Authentication Code (PAC) is incorrect:

- AUTDA, AUTDZA.
- AUTDB, AUTDZB.
- AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA.
- AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB.
- AUTIASPPC, AUTIASPPCR, AUTIA171615.
- AUTIBSPPC, AUTIBSPPCR, AUTIB171615.

If FEAT_FPACCOMBINE is implemented, the following instructions generate a PAC Fail exception when the Pointer Authentication Code (PAC) is incorrect:

- RETAA, RETAB.
- RETAASPPC, RETABSPPC.
- RETAASPPCR, RETABSPPCR.
- BLRAA, BLRAAZ, BLRAB, BLRABZ.
- BRAA, BRAB, BRAAZ, BRABZ.
- ERETAA, ERETAB.
- LDRAA, LDRAB, whether the authenticated address is written back to the base register or not.

Accessing ESR_EL2

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, ESR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0101	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ESR_EL2;
```

MSR ESR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0101	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    ESR_EL2 = X[t, 64];
```

MRS <Xt>, ESR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0010	0b000

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ESR_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ESR_EL1;
```

MSR ESR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        ESR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    ESR_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.32 FAR_EL1, Fault Address Register (EL1)

The FAR_EL1 characteristics are:

Purpose

Holds the faulting Virtual Address for all synchronous Instruction Abort exceptions, Data Abort exceptions, PC alignment fault exceptions and Watchpoint exceptions that are taken to EL1.

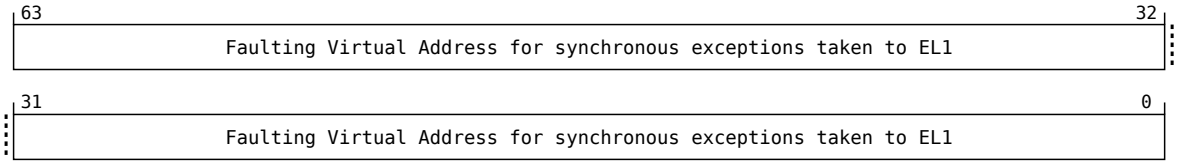
Configuration

There are no configuration notes.

Attributes

FAR_EL1 is a 64-bit register.

Field descriptions



VA, bits [63:0]

Faulting Virtual Address for synchronous exceptions taken to EL1. Exceptions that set the FAR_EL1 are Instruction Aborts (EC `0x20` or `0x21`), Data Aborts (EC `0x24` or `0x25`), PC alignment faults (EC `0x22`), and Watchpoints (EC `0x34` or `0x35`). ESR_EL1.EC holds the EC syndrome value for the exception.

For a synchronous External abort:

- If the VA that generated the abort was from an address range for which Address tagging is enabled for the translation regime in use when the abort was generated, then bits[63:56] of FAR_EL1 are UNKNOWN.
- If the VA that generated the abort was from an address range for which Address tagging is disabled and Logical Address Tagging is enabled for the translation regime in use when the abort was generated, then bits[59:56] of FAR_EL1 are UNKNOWN.

For a synchronous External abort other than a synchronous External abort on a translation table walk, this field is valid only if ESR_EL1.FnV is 0, and FAR_EL1 is UNKNOWN if ESR_EL1.FnV is 1.

If a memory fault that sets FAR_EL1 is generated from a data cache maintenance or other DC instruction, this field holds the address specified in the register argument of the instruction.

When the PE sets ESR_EL1.{FnV,FnP} to {0,1} on taking a Watchpoint exception, the PE sets FAR_EL1 to any address within the naturally-aligned fault granule that contains the virtual address of the memory access that generated the Watchpoint exception.

For a Data Abort exception or Watchpoint exception, if address tagging is enabled for the address accessed by the data access that caused the exception, then this field includes the tag. For more information about address tagging, see ‘Address tagging’.

Execution at EL0 makes FAR_EL1 become UNKNOWN.

Note

The address held in this field is an address accessed by the instruction fetch or data access that caused the exception that actually gave rise to the instruction or Data Abort. It is the lower address that gave rise to the fault that is reported. Where different faults from different addresses arise from the same instruction, such as for an instruction that loads or stores an unaligned address that crosses a page boundary, the architecture does not prioritize which fault is reported.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For all other exceptions taken to EL1, FAR_EL1 is UNKNOWN.

FAR_EL1 is made UNKNOWN on an exception return from EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing FAR_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, FAR_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0110	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = FAR_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = FAR_EL1;
```

`MSR FAR_EL1, <Xt>`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0110	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        FAR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    FAR_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.33 FAR_EL2, Fault Address Register (EL2)

The FAR_EL2 characteristics are:

Purpose

Holds the faulting Virtual Address for all synchronous Instruction Abort exceptions, Data Abort exceptions, PC alignment fault exceptions and Watchpoint exceptions that are taken to EL2.

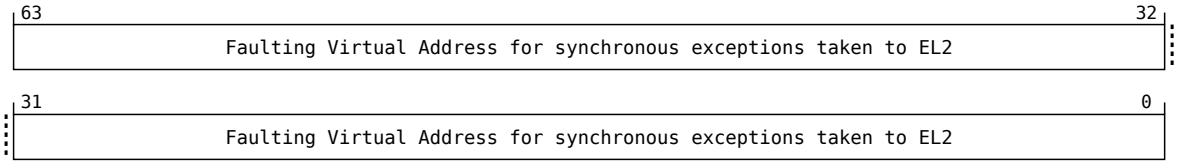
Configuration

There are no configuration notes.

Attributes

FAR_EL2 is a 64-bit register.

Field descriptions



VA, bits [63:0]

Faulting Virtual Address for synchronous exceptions taken to EL2. Exceptions that set the FAR_EL2 are Instruction Aborts (EC `0x20` or `0x21`), Data Aborts (EC `0x24` or `0x25`), PC alignment faults (EC `0x22`), and Watchpoints (EC `0x34` or `0x35`). ESR_EL2.EC holds the EC syndrome value for the exception.

For a synchronous External abort:

- If the VA that generated the abort was from an address range for which Address tagging is enabled for the translation regime in use when the abort was generated, then bits[63:56] of FAR_EL2 are UNKNOWN.
- If the VA that generated the abort was from an address range for which Address tagging is disabled and Logical Address Tagging is enabled for the translation regime in use when the abort was generated, then bits[59:56] of FAR_EL2 are UNKNOWN.

For a synchronous External abort other than a synchronous External abort on a translation table walk, this field is valid only if ESR_EL2.FnV is 0, and FAR_EL2 is UNKNOWN if ESR_EL2.FnV is 1.

If a memory fault that sets FAR_EL2 is generated from a data cache maintenance or other DC instruction, this field holds the address specified in the register argument of the instruction.

When the PE sets ESR_EL2.{FnV,FnP} to {0,1} on taking a Watchpoint exception, the PE sets FAR_EL2 to any address within the naturally-aligned fault granule that contains the virtual address of the memory access that generated the Watchpoint exception.

For a Data Abort exception or Watchpoint exception, if address tagging is enabled for the address accessed by the data access that caused the exception, then this field includes the tag. For more information about address tagging, see 'Address tagging'.

Execution at EL1 or EL0 makes FAR_EL2 become UNKNOWN.

Note

The address held in this field is an address accessed by the instruction fetch or data access that caused the exception that actually gave rise to the instruction or Data Abort. It is the lower address that gave rise to the fault that is reported. Where different faults from different addresses arise from the same instruction, such as for an instruction that loads or stores an unaligned address that crosses a page boundary, the architecture does not prioritize which fault is reported.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For all other exceptions taken to EL2, FAR_EL2 is UNKNOWN.

FAR_EL2 is made UNKNOWN on an exception return from EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing FAR_EL2

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, FAR_EL2`

op0	op1	CRn	CRm	op2
0b11	0b100	0b0110	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = FAR_EL2;
```

`MSR FAR_EL2, <Xt>`

op0	op1	CRn	CRm	op2
0b11	0b100	0b0110	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    FAR_EL2 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.34 HACR_EL2, Hypervisor Auxiliary Control Register

The HACR_EL2 characteristics are:

Purpose

Controls trapping to EL2 of IMPLEMENTATION DEFINED aspects of EL1 or EL0 operation.

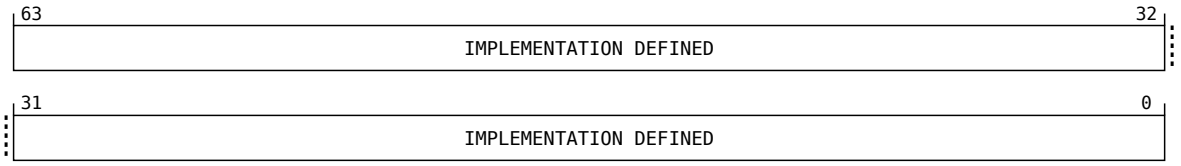
Configuration

There are no configuration notes.

Attributes

HACR_EL2 is a 64-bit register.

Field descriptions



IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing HACR_EL2

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, HACR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b111

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = HACR_EL2;
```

MSR HACR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b111

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    HACR_EL2 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.35 HCR_EL2, Hypervisor Configuration Register

The HCR_EL2 characteristics are:

Purpose

Provides configuration controls for virtualization, including defining whether various operations are trapped to EL2.

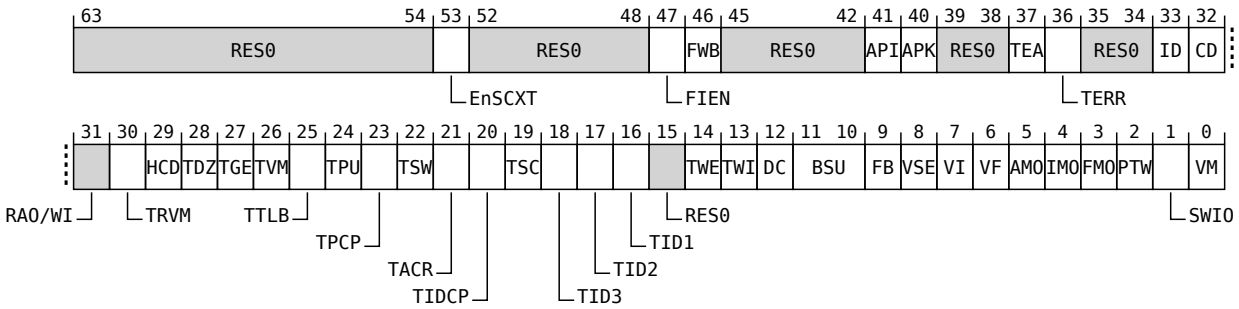
Configuration

There are no configuration notes.

Attributes

HCR_EL2 is a 64-bit register.

Field descriptions



Bits [63:54]

Reserved, RES0.

EnSCXT, bit [53]

When FEAT_CSV2_2 is implemented or FEAT_CSV2_1p2 is implemented:

Enable Access to the SCXTNUM_EL1 and SCXTNUM_EL0 registers. The defined values are:

EnSCXT	Meaning
0b0	When EL2 is enabled in the current Security state, EL1 accesses to SCXTNUM_EL0 and SCXTNUM_EL1 are disabled, causing an exception to EL2, and the value of the registers to be treated as 0. When EL2 is enabled in the current Security state, EL0 access to SCXTNUM_EL0 is disabled, causing an exception to EL2, and the value of the register to be treated as 0.
0b1	This control does not cause accesses to SCXTNUM_EL0 or SCXTNUM_EL1 to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [52:48]

Reserved, RES0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FIEN, bit [47]

When FEAT_RASv1p1 is implemented:

Fault Injection Enable. Unless this bit is set to 1, accesses to the ERXPFGCDN_EL1, ERXPFGCTL_EL1, and ERXPFGF_EL1 registers from EL1 generate a Trap exception to EL2, reported using EC syndrome value 0x18.

FIEN	Meaning
0b0	Accesses to the specified registers from EL1 are trapped to EL2.
0b1	This control does not cause any instructions to be trapped.

If ERRIDR_EL1.NUM is zero, meaning no error records are implemented, or no error record accessible using System registers is owned by a node that implements the RAS Common Fault Injection Model Extension, then this bit might be RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

FWB, bit [46]

When FEAT_S2FWB is implemented:

Forced Write-Back. Defines the combined cacheability attributes in a 2 stage translation regime.

FWB	Meaning
0b0	When this bit is 0, then the combination of stage 1 and stage 2 translations on memory type and cacheability attributes are as described in the Armv8.0 architecture. For more information, see 'Combining stage 1 and stage 2 memory type attributes'. For an R profile implementation: The encoding of the stage 2 memory type and cacheability attributes is derived from MAIR_EL2 register, as described in the Armv8-R AArch64 architecture.
0b1	When this bit is 1, then the encoding of the stage 2 memory type and cacheability attributes in bits[5:2] of the stage 2 Page or Block descriptors are as described in 'Stage 2 memory type and Cacheability attributes when FEAT_S2FWB is enabled'. For an R profile implementation: • If the inner and outer memory attributes for stage 2 EL1&0 translation regime must be the same with the same encoding, otherwise, the combined attribute is UNKNOWN. • If the stage 2 EL1&0 translation regime memory attribute is Write-Back with MAIR_EL2.Attr[7:6] = 0b11, then the combined attribute is Normal Write-Back. For all other encodings, the combination of stage 1 and stage 2 translations on memory type and cacheability attributes are as described in the Armv8.0 architecture.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [45:42]

Reserved, RES0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

API, bit [41]

When FEAT_PAuth is implemented:

Controls the use of instructions related to Pointer Authentication:

- PACGA.
- AUTDA, AUTDB, AUTDZA, AUTDZB, AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZA, AUTIZB.
- PACDA, PACDB, PACDZA, PACDZB, PACIA, PACIA1716, PACIASP, PACIAZ, PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZA, PACIZB.
- RETAA, RETAB, BRAA, BRAB, BLRAA, BLRAB, BRAAZ, BRABZ, BLRAAZ, BLRABZ.
- ERETAA, ERETAB, LDRAA, and LDRAB.

This field is ignored if the instruction is disabled as a result of the SCTLR_ELx.{EnIB, EnIA, EnDA, EnDB} fields.

API	Meaning
0b0	The instructions related to Pointer Authentication are trapped to EL2 and reported using EC syndrome value 0x09, when the instructions are enabled for the EL1&0 translation regime, from: • EL0. • EL1.
0b1	This control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

APK, bit [40]

When FEAT_PAuth is implemented:

Trap registers holding "key" values for Pointer Authentication. Traps accesses to the following registers from EL1 to EL2, reported using EC syndrome value 0x18:

- APIAKeyLo_EL1, APIAKeyHi_EL1, APIBKeyLo_EL1, APIBKeyHi_EL1, APDAKeyLo_EL1, APDAKeyHi_EL1, APDBKeyLo_EL1, APDBKeyHi_EL1, APGAKeyLo_EL1, and APGAKeyHi_EL1.

APK	Meaning
0b0	Access to the registers holding "key" values for pointer authentication from EL1 are trapped to EL2.
0b1	This control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [39:38]

Reserved, RES0.

TEA, bit [37]

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_RAS is implemented:

Route synchronous External abort exceptions to EL2.

TEA	Meaning
0b0	Synchronous External abort exceptions are unaffected by this mechanism. That is, synchronous External abort exceptions are not taken to EL2 unless routed to EL2 by another control.
0b1	When executing at Exception levels below EL2, synchronous External abort exceptions are taken to EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TERR, bit [36]

When FEAT_RAS is implemented:

Trap accesses of Error Record registers. Enables a trap to EL2 on accesses of Error Record registers.

TERR	Meaning
0b0	Accesses of the specified Error Record registers are not trapped by this mechanism.
0b1	Accesses of the specified Error Record registers at EL1 are trapped to EL2, unless the instruction generates a higher priority exception.

In AArch64 state, the instructions affected by this control are:

- MRS and MSR accesses to ERRSELR_EL1, ERXADDR_EL1, ERXCTLR_EL1, ERXMISC0_EL1, ERXMISC1_EL1, and ERXSTATUS_EL1.
- MRS accesses to ERRIDR_EL1 and ERXFR_EL1.
- If FEAT_RASv1p1 is implemented, MRS and MSR accesses to ERXMISC2_EL1 and ERXMISC3_EL1.

Unless the instruction generates a higher priority exception, trapped instructions generate an exception to EL2.

Trapped AArch64 instructions are reported using EC syndrome value 0x18.

Accessing this field has the following behavior:

- This field is permitted to be RES0 if all of the following are true:
  - ERRSELR_EL1 and all ERX* registers are implemented as UNDEFINED or RAZ/WI.
  - ERRIDR_EL1.NUM is zero.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [35:34]

Reserved, RES0.

ID, bit [33]

Stage 2 Instruction access cacheability disable. For the EL1&0 translation regime, when HCR_EL2.VM==1, this control forces all stage 2 translations for instruction accesses to Normal memory to be Non-cacheable.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

The behavior is the same irrespective of whether the instruction accesses are to an MPU region or Background region.

ID	Meaning
0b0	This control has no effect on stage 2 of the EL1&0 translation regime.
0b1	Forces all stage 2 translations for instruction accesses to Normal memory to be Non-cacheable.

This bit has no effect on the EL2 translation regime.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CD, bit [32]

Stage 2 Data access cacheability disable. For the EL1&0 translation regime, when HCR_EL2.VM==1, this control forces all stage 2 translations for data accesses and translation table walks to Normal memory to be Non-cacheable.

Note

The behavior is same irrespective of whether the data accesses is to MPU region or Background region.

CD	Meaning
0b0	This control has no effect on stage 2 of the EL1&0 translation regime for data accesses and translation table walks.
0b1	Forces all stage 2 translations for data accesses and translation table walks to Normal memory to be Non-cacheable.

This bit has no effect on the EL2 translation regime.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [31]

Reserved, RAO/WI.

TRVM, bit [30]

Trap Reads of Virtual Memory controls. Traps reads of the virtual memory control registers to EL2 as follows:

- If EL1 is using AArch64, EL1 accesses to the following registers are trapped to EL2 and reported using EC syndrome value 0x18 for MRS:
  - SCTLR_EL1, TTBR0_EL1, TTBR1_EL1, TCR_EL1, ESR_EL1, FAR_EL1, AFSR0_EL1, AFSR1_EL1, MAIR_EL1, AMAIR_EL1, and CONTEXTIDR_EL1.
  - If EL1 is in PMSAv8-64 context, EL1 accesses to the following registers are also trapped to EL2 and reported using EC syndrome value 0x18 - PRENR_EL1, PRSELR_EL1, PRBAR_EL1, PRBARn_EL1, PRLAR_EL1, and PRLARn_EL1.

TRVM	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	Read accesses to the specified Virtual Memory control registers are trapped to EL2.

When HCR_EL2.TGE is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

EL2 provides a second stage of address translation, that a hypervisor can use to remap the address map defined by a Guest OS. In addition, a hypervisor can trap attempts by a Guest OS to write to the registers that control the memory system. A hypervisor might use this trap as part of its virtualization of memory management.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

HCD, bit [29]

HVC instruction disable. Disables EL1 and EL2 execution of HVC instructions, from both Execution states, reported using EC syndrome value `0x00`.

HCD	Meaning
`0b0`	HVC instruction execution is enabled at EL2 and EL1.
`0b1`	HVC instructions are UNDEFINED at EL2 and EL1. Any resulting exception is taken to the Exception level at which the HVC instruction is executed.

Note

HVC instructions are always UNDEFINED at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TDZ, bit [28]

Traps EL0 and EL1 execution of the following instructions to EL2 from AArch64 state only, reported using EC syndrome value `0x18`:

- DC ZVA.

TDZ	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	In AArch64 state, any attempt to execute an instruction this trap applies to at EL1, or at EL0 when the instruction is not UNDEFINED at EL0, is trapped to EL2. Reading the DCZID_EL0 returns a value that indicates that the instructions this trap applies to are not supported.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TGE, bit [27]

Trap General Exceptions, from EL0.

TGE	Meaning
`0b0`	This control has no effect on execution at EL0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TGE	Meaning
0b1	When EL2 is not enabled in the current Security state, this control has no effect on execution at EL0. When EL2 is enabled in the current Security state, in all cases: • All exceptions that would be routed to EL1 are routed to EL2. • If EL1 is using AArch64, the SCTLR_EL1.M field is treated as being 0 for all purposes other than returning the result of a direct read of SCTLR_EL1. • If stage 1 EL1&0 translation regime is in PMSAv8-64 context, the SCTLR_EL1.BR field is treated as being 0 for all purposes other than returning the result of a direct read of SCTLR_EL1. • All virtual interrupts and virtual exceptions are disabled. • Any IMPLEMENTATION DEFINED mechanisms for signaling virtual interrupts are disabled. • An exception return to EL1 is treated as an illegal exception return. • The MDCR_EL2.{TDRA, TDOSA, TDA, TDE} fields are treated as being 1 for all purposes other than returning the result of a direct read of MDCR_EL2. In addition, when EL2 is enabled in the current Security state, the Effective values of the HCR_EL2.{FMO, IMO, AMO} fields are 1.

HCR_EL2.TGE must not be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TVM, bit [26]

Trap Virtual Memory controls. Traps writes to the virtual memory control registers to EL2 as follows:

- If EL1 is using AArch64, the following registers are trapped to EL2 and reported using EC syndrome value 0x18 for MSR:
  - SCTLR_EL1, TTBR0_EL1, TTBR1_EL1, TCR_EL1, ESR_EL1, FAR_EL1, AFSR0_EL1, AFSR1_EL1, MAIR_EL1, AMAIR_EL1, and CONTEXTIDR_EL1.
  - If EL1 is in PMSAv8-64 context, EL1 accesses to the following registers are also trapped to EL2 and reported using EC syndrome value 0x18 - PRENR_EL1, PRSELR_EL1, PRBAR_EL1, PRBARn_EL1, and PRLAR_EL1, and PRLARn_EL1.

TVM	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	Write accesses to the specified Virtual Memory control registers are trapped to EL2.

When HCR_EL2.TGE is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TTLB, bit [25]

Trap TLB maintenance instructions. Traps execution of TLB maintenance instructions at EL1 to EL2 as follows:

- If EL1 is using AArch64, then the following instructions are trapped to EL2 and reported using EC syndrome value 0x18:
  - TLBI VMALLE1, TLBI VAE1, TLBI ASIDE1, TLBI VAAE1, TLBI VALE1, and TLBI VAALE1.
  - TLBI VMALLE1IS, TLBI VAE1IS, TLBI ASIDE1IS, TLBI VAAE1IS, TLBI VALE1IS, and TLBI VAALE1IS.
  - If FEAT_TLBIOS is implemented, TLBI VMALLE1OS, TLBI VAE1OS, TLBI ASIDE1OS, TLBI VAAE1OS, TLBI VALE1OS, and TLBI VAALE1OS.
  - If FEAT_TLBIRANGE is implemented, TLBI RVAE1, TLBI RVAAE1, TLBI RVALE1, TLBI RVAALE1, TLBI RVAE1IS, TLBI RVAAE1IS, TLBI RVALE1IS, and TLBI RVAALE1IS.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If FEAT_TLBIOS and FEAT_TLBIRANGE are implemented, TLBI RVAE1OS, TLBI RVAAE1OS, TLBI RVALE1OS, and TLBI RVAALE1OS.

TTLB	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	EL1 execution of the specified TLB maintenance instructions are trapped to EL2.

When HCR_EL2.TGE is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

Note

The TLB maintenance instructions are UNDEFINED at EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TPU, bit [24]

Trap cache maintenance instructions that operate to the Point of Unification. Traps execution of those cache maintenance instructions at EL0 and EL1 to EL2 as follows:

- If EL0 is using AArch64 and the value of SCTLR_EL1.UCI is 1, then the following instructions at EL0 are trapped to EL2 and reported with EC syndrome value 0x18:
  - IC IVAU and DC CVAU.
- If EL1 is using AArch64, then the following instructions at EL1 are trapped to EL2 and reported with EC syndrome value 0x18:
  - IC IVAU, IC IALLU, IC IALLUIS, and DC CVAU.

Note

When SCTLR_EL1.UCI is 0, the trap on execution of instructions at EL0 is higher priority than this control. An exception generated because an instruction is UNDEFINED at EL0 is higher priority than this trap to EL2. In addition:

- IC IALLUIS and IC IALLU are always UNDEFINED at EL0 using AArch64.

TPU	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	For each of the specified instructions, if the execution of the instruction can be trapped, access is trapped to EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TPCP, bit [23]

Trap data or unified cache maintenance instructions that operate to the Point of Coherency or Persistence. When EL2 is enabled in the current Security state, traps execution of cache maintenance instructions at EL0 and EL1 to EL2 as follows:

- If EL0 is using AArch64 and the value of SCTLR_EL1.UCI is 1, then the following instructions at EL0 are trapped to EL2 and reported using EC syndrome value 0x18:
  - DC CIVAC and DC CVAC.
  - If FEAT_DPB is implemented, DC CVAP.
  - If FEAT_DPB2 is implemented, DC CVADP.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If EL1 is using AArch64, then the following instructions at EL1 are trapped to EL2 and reported using EC syndrome value `0x18`:
  - DC IVAC, DC CIVAC and DC CVAC.
  - If FEAT_DPB is implemented, DC CVAP.
  - If FEAT_DPB2 is implemented, DC CVADP.

Note

This field was previously named TPC.
When SCTLR_EL1.UCI is 0, the trap on execution of instructions at EL0 is higher priority than this control.
An exception generated because an instruction is UNDEFINED at EL0 is higher priority than this trap to EL2. In addition:

- AArch64 instructions which invalidate by VA to the Point of Coherency are always UNDEFINED at EL0 using AArch64.

TPCP	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	For each of the specified instructions, if the execution of the instruction can be trapped, it is trapped to EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TSW, bit [22]

Trap data or unified cache maintenance instructions that operate by Set/Way. Traps execution of those cache maintenance instructions at EL1 to EL2 as follows:

- If EL1 is using AArch64, then the following instructions are trapped to EL2 and reported with EC syndrome value `0x18`:
  - DC ISW, DC CSW, and DC CISW.

Note

An exception generated because an instruction is UNDEFINED at EL0 is higher priority than this trap to EL2, and these instructions are always UNDEFINED at EL0.

TSW	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	Execution of the specified instructions is trapped to EL2.

When HCR_EL2.TGE is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TACR, bit [21]

Trap Auxiliary Control Registers. Traps EL1 accesses to the Auxiliary Control Registers to EL2 as follows:

- If EL1 is using AArch64, accesses to ACTLR_EL1 to EL2, are trapped to EL2 and reported using EC syndrome value `0x18`.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TACR	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	EL1 accesses to the specified registers are trapped to EL2.

When HCR_EL2.TGE is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

Note

ACTLR_EL1 is not accessible at EL0.
The Auxiliary Control Registers are IMPLEMENTATION DEFINED registers that might implement global control bits for the PE.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TIDCP, bit [20]

Trap IMPLEMENTATION DEFINED functionality. Traps EL1 accesses to the encodings reserved for IMPLEMENTATION DEFINED functionality to EL2 as follows:

- In AArch64 state, EL0 access to the encodings in the following reserved encoding spaces are trapped:
  - IMPLEMENTATION DEFINED System instructions, which are accessed using SYS and SYSL, with CRn == {11, 15}, and are reported using EC syndrome value 0x18.
  - IMPLEMENTATION DEFINED System instructions, which are accessed using SYSP, with CRn == {11, 15}, and are reported using EC syndrome value 0x14.
  - IMPLEMENTATION DEFINED System registers, which are accessed using MRS and MSR with the S3_<op1>_<Cn>_<Cm>_<op2> register name, and are reported using EC syndrome 0x18.
  - IMPLEMENTATION DEFINED System registers, which are accessed using MRRS and MSRR with the S3_<op1>_<Cn>_<Cm>_<op2> register name, and are reported using EC syndrome 0x14.

When this functionality is accessed from EL0:

- If HCR_EL2.TIDCP is 1, it is IMPLEMENTATION DEFINED whether any accesses from EL0 are trapped to EL2.
- If HCR_EL2.TIDCP is 0, any accesses from EL0 are UNDEFINED and generate an exception that is taken to EL1 or EL2.

TIDCP	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	EL1 accesses to or execution of the specified encodings reserved for IMPLEMENTATION DEFINED functionality are trapped to EL2.

An implementation can also include IMPLEMENTATION DEFINED registers that provide additional controls, to give finer-grained control of the trapping of IMPLEMENTATION DEFINED features.

Note

The trapping of accesses to these registers from EL1 is higher priority than an exception resulting from the register access being UNDEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TSC, bit [19]

Trap SMC instructions. Traps EL1 execution of SMC instructions to EL2.

If execution is in AArch64 state, the trap is reported using EC syndrome value 0x17.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

HCR_EL2.TSC traps execution of the SMC instruction. It is not a routing control for the SMC exception. Trap exceptions and SMC exceptions have different preferred return addresses.

TSC	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	If EL3 is not implemented and the Effective value of HCR_EL2.NV is 1, then any attempt to execute an SMC instruction at EL1 using AArch64 is trapped to EL2. If EL3 is not implemented and the Effective value of HCR_EL2.NV is 0, then it is IMPLEMENTATION DEFINED whether: • Any attempt to execute an SMC instruction at EL1 is trapped to EL2. • Any attempt to execute an SMC instruction is UNDEFINED.

SMC instructions are UNDEFINED at EL0.

If EL3 is not implemented, and the Effective value of HCR_EL2.NV is 0, then it is IMPLEMENTATION DEFINED whether this bit is:

- RES0.
- Implemented with the functionality as described in HCR_EL2.TSC.

When HCR_EL2.TGE is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TID3, bit [18]

Trap ID group 3. Traps EL1 reads of group 3 ID registers to EL2, when EL2 is enabled in the current Security state, as follows:

In AArch64 state:

- Reads of the following registers are trapped to EL2, reported using EC syndrome value 0x18:
  - ID_PFR0_EL1, ID_PFR1_EL1, ID_DFR0_EL1, ID_AFR0_EL1, ID_MMFR0_EL1, ID_MMFR1_EL1, ID_MMFR2_EL1, ID_MMFR3_EL1, ID_ISAR0_EL1, ID_ISAR1_EL1, ID_ISAR2_EL1, ID_ISAR3_EL1, ID_ISAR4_EL1, ID_ISAR5_EL1, MVFR0_EL1, MVFR1_EL1, and MVFR2_EL1.
  - ID_AA64PFR0_EL1, ID_AA64PFR1_EL1, ID_AA64DFR0_EL1, ID_AA64DFR1_EL1, ID_AA64ISAR0_EL1, ID_AA64ISAR1_EL1, ID_AA64MMFR0_EL1, ID_AA64MMFR1_EL1, ID_AA64AFR0_EL1, and ID_AA64AFR1_EL1.
  - Reads of the following registers are trapped to EL2, unless the register is implemented as RAZ when it it IMPLEMENTATION DEFINED whether accesses are trapped to EL2.
  - — ID_PFR2_EL1, ID_MMFR4_EL1 and ID_MMFR5_EL1.
  - — ID_AA64MMFR4_EL1.
  - — ID_AA64MMFR2_EL1 and ID_ISAR6_EL1.
  - It is IMPLEMENTATION DEFINED whether this field traps MRS accesses to registers in the following range that are not already mentioned in this field description: op0 == 3, op1 == 0, CRn == 0, CRm == {2-7}, op2 == {0-7}.

TID3	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	The specified EL1 read accesses to ID group 3 registers are trapped to EL2.

When HCR_EL2.TGE is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

The reset behavior of this field is:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TID2, bit [17]

Trap ID group 2. Traps the following register accesses to EL2 as follows:

- If EL1 is using AArch64, reads of CTR_EL0, CCSIDR_EL1, CCSIDR2_EL1, CLIDR_EL1, and CSSELR_EL1 are trapped to EL2, reported using EC syndrome value `0x18`.
- If EL0 is using AArch64 and the value of SCTLR_EL1.UCT is not 0, reads of CTR_EL0 are trapped to EL2, reported using EC syndrome value `0x18`. If the value of SCTLR_EL1.UCT is 0, then EL0 reads of CTR_EL0 are trapped to EL1 and the resulting exception takes precedence over this trap.
- If EL1 is using AArch64, writes to CSSELR_EL1 are trapped to EL2, reported using EC syndrome value `0x18`.

TID2	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	The specified EL1 and EL0 accesses to ID group 2 registers are trapped to EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TID1, bit [16]

Trap ID group 1. Traps EL1 reads of the following registers to EL2 as follows:

- Accesses of MPUIR_EL1, REVIDR_EL1, and AIDR_EL1, reported using EC syndrome value `0x18`.

TID1	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	The specified EL1 read accesses to ID group 1 registers are trapped to EL2.

When HCR_EL2.TGE is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [15]

Reserved, RES0.

TWE, bit [14]

Traps EL0 and EL1 execution of WFE instructions to EL2 from both Execution states, reported using EC syndrome value `0x01`.

TWE	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	Any attempt to execute a WFE instruction at EL0 or EL1 is trapped to EL2, if the instruction would otherwise have caused the PE to enter a low-power state and it is not trapped by SCTLR.nTWE or SCTLR_EL1.nTWE.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Since a WFE can complete at any time, even without a Wakeup event, the traps on WFE are not guaranteed to be taken, even if the WFE is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

For more information about when WFE instructions can cause the PE to enter a low-power state, see 'Wait for Event mechanism and Send event'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TWI, bit [13]

Traps EL0 and EL1 execution of WFI instructions to EL2 from both Execution states, reported using EC syndrome value `0x01`.

TWI	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	Any attempt to execute a WFI instruction at EL0 or EL1 is trapped to EL2, if the instruction would otherwise have caused the PE to enter a low-power state and it is not trapped by SCTLR.nTWI or SCTLR_EL1.nTWI.

Note

Since a WFI can complete at any time, even without a Wakeup event, the traps on WFI are not guaranteed to be taken, even if the WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

For more information about when WFI instructions can cause the PE to enter a low-power state, see 'Wait for Interrupt'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

DC, bit [12]

Default Cacheability.

DC	Meaning
`0b0`	This control has no effect on the EL1&0 translation regime.
`0b1`	In both Security states: • When EL1 is using AArch64, the PE behaves as if the value of the SCTLR_EL1.M field is 0 for all purposes other than returning the value of a direct read of SCTLR_EL1. • If stage 1 EL1&0 translation regime is in PMSAv8-64 context, the PE behaves as if the value of the SCTLR_EL1.BR field is 0 for all purposes other than returning the value of a direct read of SCTLR_EL1. • The PE behaves as if the value of the HCR_EL2.VM field is 1 for all purposes other than returning the value of a direct read of HCR_EL2. • The memory type produced by stage 1 of the EL10 translation regime is Normal Non-Shareable, Inner Write-Back Read-Allocate Write-Allocate, Outer Write-Back Read-Allocate Write-Allocate.

This field has no effect on the EL2 translation regime.

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

BSU, bits [11:10]

Barrier Shareability upgrade. This field determines the minimum shareability domain that is applied to any barrier instruction executed from EL1 or EL0:

BSU	Meaning
0b00	No effect.
0b01	Inner Shareable.
0b10	Outer Shareable.
0b11	Full system.

This value is combined with the specified level of the barrier held in its instruction, using the same principles as combining the shareability attributes from two stages of address translation.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

FB, bit [9]

Force broadcast. Causes the following Non-shareable invalidate instructions to be broadcast within the Inner Shareable domain when executed from EL1:

- In AArch64 state:
  - All TLBI instructions that are executable at EL1 and do not specify IS or OS.
  - IC IALLU.

For more information, see 'A64 System instructions for TLB maintenance'.

FB	Meaning
0b0	The specified instructions are not affected by this control.
0b1	When one of the specified Non-shareable instructions is executed at EL1, the operation is broadcast within the Inner Shareable shareability domain.

When HCR_EL2.TGE is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

VSE, bit [8]

Virtual SError exception.

VSE	Meaning
0b0	This mechanism is not making a virtual SError exception pending.
0b1	A virtual SError exception is pending because of this mechanism.

The virtual SError exception is enabled only when HCR_EL2.{TGE, AMO} is {0, 1}.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

VI, bit [7]

Virtual IRQ Interrupt.

VI	Meaning
0b0	This mechanism is not making a virtual IRQ pending.
0b1	A virtual IRQ is pending because of this mechanism.

The virtual IRQ is enabled only when the value of HCR_EL2.{TGE, IMO} is {0, 1}.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

VF, bit [6]

Virtual FIQ Interrupt.

VF	Meaning
0b0	This mechanism is not making a virtual FIQ pending.
0b1	A virtual FIQ is pending because of this mechanism.

The virtual FIQ is enabled only when the value of HCR_EL2.{TGE, FMO} is {0, 1}.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

AMO, bit [5]

Physical SError exception routing.

AMO	Meaning
0b0	When executing at Exception levels below EL2, and EL2 is enabled in the current Security state, all of the following apply: • If HCR_EL2.TGE is 0, then the routing of physical SError exceptions is unaffected by this mechanism • If HCR_EL2.TGE is 1, then physical SError exceptions are taken to EL2. • Virtual SError exceptions are disabled.
0b1	When the Effective value of HCR_EL2.TGE is 0: • Physical SError exceptions are taken to EL2. • Virtual SError exceptions are enabled.

When the Effective value of HCR_EL2.TGE is 1, regardless of the value of the AMO bit, all of the following are true:

- Physical SError exceptions target EL2.
- Virtual SError exceptions are disabled.

When the Effective value of HCR_EL2.TGE is 0, it is IMPLEMENTATION DEFINED whether the Effective value of HCR_EL2.AMO is 1 or the value programmed.

For more information, see 'Asynchronous exception routing'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IMO, bit [4]

Physical IRQ Routing.

IMO	Meaning
0b0	When the Effective value of HCR_EL2.TGE is 0: • Physical IRQ interrupts are not taken to EL2. • Virtual IRQ interrupts are disabled.
0b1	When the Effective value of HCR_EL2.TGE is 0: • Physical IRQ exceptions are taken to EL2. • Virtual IRQ interrupts are enabled.

When the Effective value of HCR_EL2.TGE is 1, regardless of the value of the IMO bit, all of the following are true:

- Physical IRQ Interrupts target EL2 unless they are routed to EL3.
- Virtual IRQ interrupts are disabled.

When the Effective value of HCR_EL2.TGE is 0, it is IMPLEMENTATION DEFINED whether the Effective value of HCR_EL2.IMO is 1 or the value programmed.

For more information, see 'Asynchronous exception routing'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

FMO, bit [3]

Physical FIQ Routing.

FMO	Meaning
0b0	When the Effective value of HCR_EL2.TGE is 0: • Physical FIQ interrupts are not taken to EL2. • Virtual FIQ interrupts are disabled.
0b1	When the Effective value of HCR_EL2.TGE is 0: • Physical FIQ exceptions are taken to EL2. • Virtual FIQ interrupts are enabled.

When the Effective value of HCR_EL2.TGE is 1, regardless of the value of the FMO bit, all of the following are true:

- Physical FIQ Interrupts target EL2 unless they are routed to EL3.
- Virtual FIQ interrupts are disabled.

When the Effective value of HCR_EL2.TGE is 0, it is IMPLEMENTATION DEFINED whether the Effective value of HCR_EL2.FMO is 1 or the value programmed.

For more information, see 'Asynchronous exception routing'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PTW, bit [2]

Protected Table Walk. In the EL1&0 translation regime, a translation table access made as part of a stage 1 translation table walk is subject to a stage 2 translation. The combining of the memory type attributes from the two

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

stages of translation means the access might be made to a type of Device memory. If this occurs, then the value of this bit determines the behavior:

PTW	Meaning
0b0	The translation table walk occurs as if it is to Normal Non-cacheable memory. This means it can be made speculatively.
0b1	The memory access generates a stage 2 Permission fault.

This bit is permitted to be cached in a TLB.

When HCR_EL2.TGE is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

In a PMSA-only implementation, this bit is permitted to be RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SWIO, bit [1]

Set/Way Invalidation Override. Causes EL1 execution of the data cache invalidate by set/way instructions to perform a data cache clean and invalidate by set/way:

SWIO	Meaning
0b0	This control has no effect on the operation of data cache invalidate by set/way instructions.
0b1	Data cache invalidate by set/way instructions perform a data cache clean and invalidate by set/way.

When the value of this bit is 1:

AArch64: DC ISW performs the same invalidation as a DC CISW instruction.

This bit can be implemented as RES1.

When HCR_EL2.TGE is 1, the PE ignores the value of this field for all purposes other than a direct read of this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

VM, bit [0]

Virtualization enable. Enables stage 2 address translation for the EL1&0 translation regime.

VM	Meaning
0b0	EL1&0 stage 2 address translation disabled.
0b1	EL1&0 stage 2 address translation enabled.

If HCR_EL2.VM is 1 and SCTLR_EL2.{M, BR} is {0, 0}, then the behavior is a CONSTRAINED UNPREDICTABLE choice of:

- The memory attribute becomes UNKNOWN.
- Raise stage 2 level 0 Translation fault.

When the value of this bit is 1, data cache invalidate instructions executed at EL1 perform a data cache clean and invalidate. For the invalidate by set/way instruction this behavior applies regardless of the value of the HCR_EL2.SWIO bit.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing HCR_EL2

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, HCR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = HCR_EL2;
```

MSR HCR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    HCR_EL2 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.36 HPFAR_EL2, Hypervisor IPA Fault Address Register

The HPFAR_EL2 characteristics are:

Purpose

Holds the faulting IPA for some aborts on a stage 2 translation taken to EL2 due to a fault on an access for a stage 2 translation.

Configuration

The HPFAR_EL2 is written for:

- A Translation fault, Access Flag fault, or Address Size fault on a stage 2 translation not on a stage 1 translation table walk.
- A Translation fault, Access Flag fault, Address Size fault, or Permission fault on stage 2 translation of an address accessed in a stage 1 translation table walk.

For all other exceptions taken to EL2, this register is UNKNOWN.

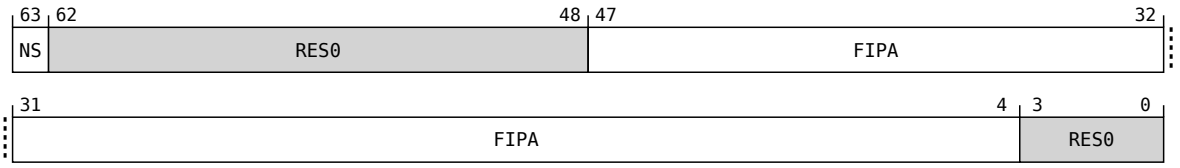
Note

The address held in this register is an address accessed by the instruction fetch or data access that caused the exception that gave rise to the Instruction Abort exception or Data Abort exception. It is the lower address that gave rise to the fault that is reported. Where different faults from different addresses arise from the same instruction, such as for an instruction that loads or stores an unaligned address that crosses a page boundary, the architecture does not prioritize which fault is reported.

Attributes

HPFAR_EL2 is a 64-bit register.

Field descriptions



Execution at EL1 or EL0 makes HPFAR_EL2 become UNKNOWN.

NS, bit [63]

When FEAT_SEL2 is implemented:

Faulting IPA address space.

NS	Meaning
0b0	Faulting IPA is from the Secure IPA space.
0b1	Faulting IPA is from the Non-secure IPA space.

For Data Abort exceptions or Instruction Abort exceptions taken to Non-secure EL2:

- This field is RES0.
- The address is from the Non-secure IPA space.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

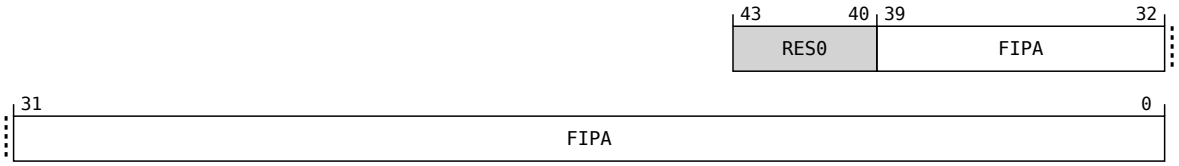
Reserved, RES0.

Bits [62:48]

Reserved, RES0.

FIPA, bits [47:4]

When FEAT_LPA is implemented:



Bits [43:40]

Reserved, RES0.

FIPA, bits [39:0]

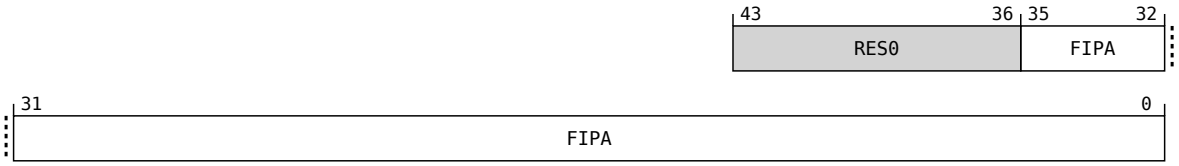
Bits [51:12] of the Faulting Intermediate Physical Address.

For implementations with fewer than 52 physical address bits, the corresponding upper bits in this field are RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_LPA is not implemented:



Bits [43:36]

Reserved, RES0.

FIPA, bits [35:0]

Bits[47:12] Faulting Intermediate Physical Address.

For implementations with fewer than 48 physical address bits, the corresponding upper bits in this field are RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [3:0]

Reserved, RES0.

Accessing HPFAR_EL2

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, HPFAR_EL2`

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b100	0b0110	0b0000	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = HPFAR_EL2;
```

MSR HPFAR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0110	0b0000	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    HPFAR_EL2 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.37 HSTR_EL2, Hypervisor System Trap Register

The HSTR_EL2 characteristics are:

Purpose

Controls trapping to EL2 of EL1 or lower AArch32 accesses to the System register in the coproc == 0b1111 encoding space, by the CRn value used to access the register using MCR or MRC instruction. When the register is accessible using an MCRR or MRRC instruction, this is the CRm value used to access the register.

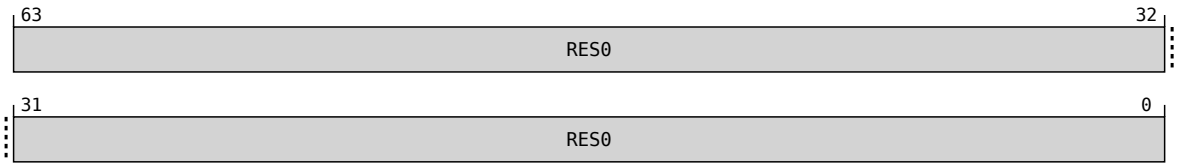
Configuration

There are no configuration notes.

Attributes

HSTR_EL2 is a 64-bit register.

Field descriptions



Bits [63:0]

Reserved, RES0.

Accessing HSTR_EL2

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, HSTR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = HSTR_EL2;
```

MSR HSTR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    HSTR_EL2 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.38 ID_AA64AFR0_EL1, AArch64 Auxiliary Feature Register 0

The ID_AA64AFR0_EL1 characteristics are:

Purpose

Provides information about the IMPLEMENTATION DEFINED features of the PE in AArch64 state.

For general information about the interpretation of the ID registers, see 'Principles of the ID scheme for fields in ID registers'.

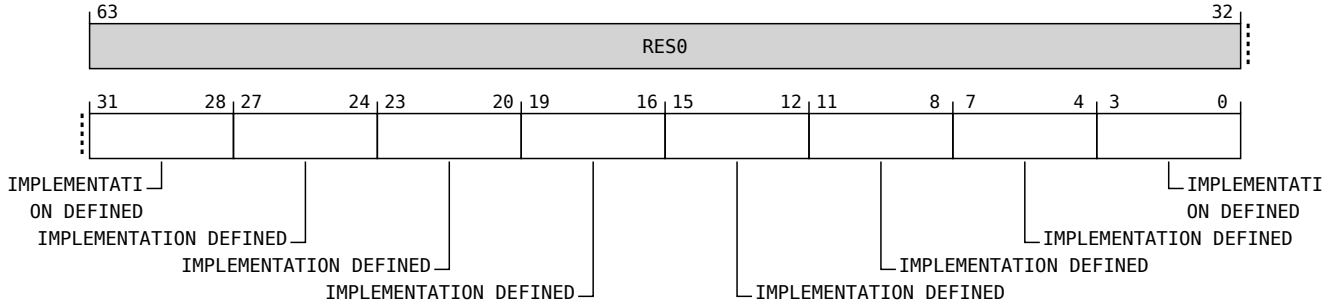
Configuration

There are no configuration notes.

Attributes

ID_AA64AFR0_EL1 is a 64-bit register.

Field descriptions



Bits [63:32]

Reserved, RES0.

IMPLEMENTATION DEFINED, bits [31:28]

IMPLEMENTATION DEFINED.

IMPLEMENTATION DEFINED, bits [27:24]

IMPLEMENTATION DEFINED.

IMPLEMENTATION DEFINED, bits [23:20]

IMPLEMENTATION DEFINED.

IMPLEMENTATION DEFINED, bits [19:16]

IMPLEMENTATION DEFINED.

IMPLEMENTATION DEFINED, bits [15:12]

IMPLEMENTATION DEFINED.

IMPLEMENTATION DEFINED, bits [11:8]

IMPLEMENTATION DEFINED.

IMPLEMENTATION DEFINED, bits [7:4]

IMPLEMENTATION DEFINED.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IMPLEMENTATION DEFINED, bits [3:0]

IMPLEMENTATION DEFINED.

Accessing ID_AA64AFR0_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, ID_AA64AFR0_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0101	0b100

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ID_AA64AFR0_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ID_AA64AFR0_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.39 ID_AA64AFR1_EL1, AArch64 Auxiliary Feature Register 1

The ID_AA64AFR1_EL1 characteristics are:

Purpose

Reserved for future expansion of information about the IMPLEMENTATION DEFINED features of the PE in AArch64 state.

For general information about the interpretation of the ID registers, see 'Principles of the ID scheme for fields in ID registers'.

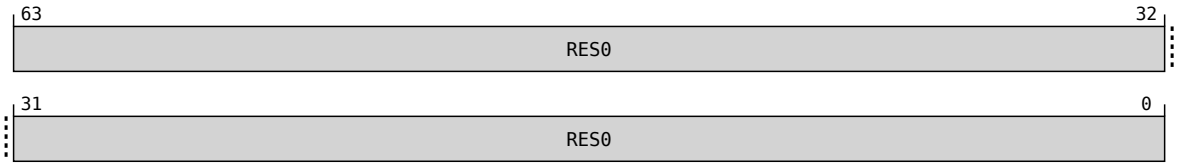
Configuration

There are no configuration notes.

Attributes

ID_AA64AFR1_EL1 is a 64-bit register.

Field descriptions



Bits [63:0]

Reserved, RES0.

Accessing ID_AA64AFR1_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ID_AA64AFR1_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0101	0b101

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ID_AA64AFR1_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ID_AA64AFR1_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.40 ID_AA64DFR0_EL1, AArch64 Debug Feature Register 0

The ID_AA64DFR0_EL1 characteristics are:

Purpose

Provides top-level information about the debug system in AArch64 state.

For general information about the interpretation of the ID registers, see 'Principles of the ID scheme for fields in ID registers'.

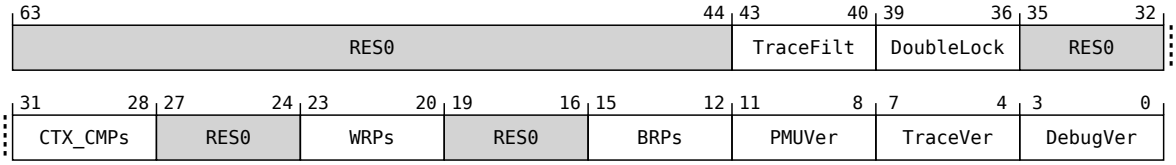
Configuration

The external register EDDFR gives information from this register.

Attributes

ID_AA64DFR0_EL1 is a 64-bit register.

Field descriptions



Bits [63:44]

Reserved, RES0.

TraceFilt, bits [43:40]

Armv8.4 Self-hosted Trace Extension version.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TraceFilt	Meaning
0b0000	Armv8.4 Self-hosted Trace Extension not implemented.
0b0001	Armv8.4 Self-hosted Trace Extension implemented.

All other values are reserved.

FEAT_TRF implements the functionality identified by the value 0b0001.

From Armv8.4, if FEAT_ETMv4 is implemented, the value 0b0000 is not permitted.

Access to this field is RO.

DoubleLock, bits [39:36]

OS Double Lock implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

DoubleLock	Meaning
0b0000	OS Double Lock implemented. OSDLR_EL1 is RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b1111	OS Double Lock not implemented. OSDLR_EL1 is RAZ/WI.

All other values are reserved.

FEAT_DoubleLock implements the functionality identified by the value 0b0000.

In Armv8.0, the only permitted value is 0b0000.

If FEAT_Debugv8p2 is implemented and FEAT_DoPD is not implemented, the permitted values are 0b0000 and 0b1111.

If FEAT_DoPD is implemented, the only permitted value is 0b1111.

Access to this field is RO.

Bits [35:32]

Reserved, RES0.

CTX_CMPs, bits [31:28]

Number of context-aware breakpoints, minus 1.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CTX_CMPs	Meaning
0b0000..0b1110	The number of context-aware breakpoints, minus 1.
0b1111	If ID_AA64DFR1_EL1.CTX_CMPs is zero, then 16 context-aware breakpoints are implemented. Otherwise, 16 or more context-aware breakpoints are implemented and ID_AA64DFR1_EL1.CTX_CMPs indicates how many.

Values greater than ID_AA64DFR0_EL1.BRPs are not permitted.

Access to this field is RO.

Bits [27:24]

Reserved, RES0.

WRPs, bits [23:20]

Number of watchpoints, minus 1.

The value of this field is an IMPLEMENTATION DEFINED choice of:

WRPs	Meaning
0b0001..0b1111	The number of watchpoints, minus 1.

The value 0b0000 is reserved.

Access to this field is RO.

Bits [19:16]

Reserved, RES0.

BRPs, bits [15:12]

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Index of the highest numbered context-aware breakpoint. Identifies which of the implemented breakpoints are context-aware breakpoints.

The value of this field is an IMPLEMENTATION DEFINED choice of:

BRPs	Meaning
0b0001..0b1110	Index of the highest numbered context-aware breakpoint.
0b1111	If ID_AA64DFR0_EL1.CTX_CMPs and ID_AA64DFR1_EL1.CTX_CMPs indicate that 16 or fewer breakpoints are context-aware, then the index of the highest context-aware breakpoint is 15. Otherwise, the context-aware breakpoints are the lowest numbered breakpoints.

The value 0b0000 is reserved.

Access to this field is RO.

PMUVer, bits [11:8]

Performance Monitors Extension version.

This field does not follow the standard ID scheme, but uses the alternative ID scheme described in 'Alternative ID scheme used for the Performance Monitors Extension version'.

The value of this field is an IMPLEMENTATION DEFINED choice of:

All other values are reserved.

FEAT_PMUv3 implements the functionality identified by the value 0b0001.

FEAT_PMUv3p1 implements the functionality identified by the value 0b0100.

FEAT_PMUv3p4 implements the functionality identified by the value 0b0101.

From Armv8.1, if FEAT_PMUv3 is implemented, the value 0b0001 is not permitted.

From Armv8.4, if FEAT_PMUv3 is implemented, the value 0b0100 is not permitted.

Access to this field is RO.

TraceVer, bits [7:4]

Trace support. Indicates whether System register interface to a trace unit is implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TraceVer	Meaning
0b0000	Trace unit System registers not implemented.
0b0001	Trace unit System registers implemented.

All other values are reserved.

See the ETM Architecture Specification for more information.

A value of 0b0000 only indicates that no System register interface to a trace unit is implemented. A trace unit might nevertheless be implemented without a System register interface.

Access to this field is RO.

DebugVer, bits [3:0]

Debug architecture version. Indicates presence of Armv8 debug architecture.

The value of this field is an IMPLEMENTATION DEFINED choice of:

All other values are reserved.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_Debugv8p1 implements the functionality identified by the value 0b0111.

FEAT_Debugv8p2 implements the functionality identified by the value 0b1000.

FEAT_Debugv8p4 implements the functionality identified by the value 0b1001.

From Armv8.1, when FEAT_Debugv8p1 is implemented the value 0b0110 is not permitted.

From Armv8.2, the values 0b0110 and 0b0111 are not permitted.

From Armv8.4, the value 0b1000 is not permitted.

Access to this field is RO.

Accessing ID_AA64DFR0_EL1

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, ID_AA64DFR0_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0101	0b000

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ID_AA64DFR0_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ID_AA64DFR0_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.41 ID_AA64DFR1_EL1, AArch64 Debug Feature Register 1

The ID_AA64DFR1_EL1 characteristics are:

Purpose

Provides top-level information about the debug system in AArch64.

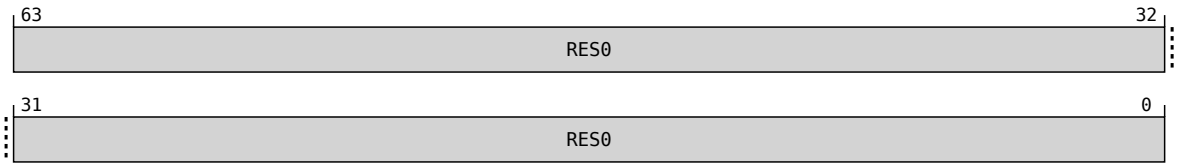
Configuration

There are no configuration notes.

Attributes

ID_AA64DFR1_EL1 is a 64-bit register.

Field descriptions



Bits [63:0]

Reserved, RES0.

Accessing ID_AA64DFR1_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, ID_AA64DFR1_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0101	0b001

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ID_AA64DFR1_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ID_AA64DFR1_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.42 ID_AA64DFR2_EL1, AArch64 Debug Feature Register 2

The ID_AA64DFR2_EL1 characteristics are:

Purpose

Provides top-level information about the debug system in AArch64.

For general information about the interpretation of the ID registers, see 'Principles of the ID scheme for fields in ID registers'.

Configuration

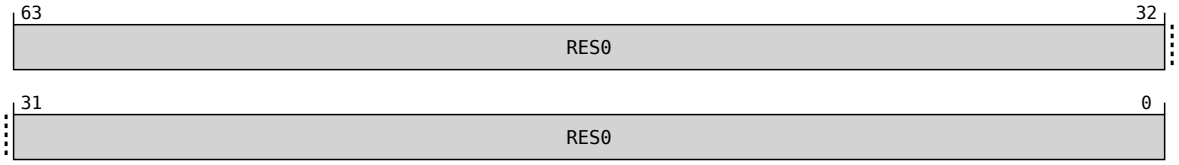
Note

Prior to the introduction of the features described by this register, this register was unnamed and reserved, RES0 from EL1 and EL2.

Attributes

ID_AA64DFR2_EL1 is a 64-bit register.

Field descriptions



Bits [63:0]

Reserved, RES0.

Accessing ID_AA64DFR2_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, ID_AA64DFR2_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0101	0b010

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if (!IsZero(ID_AA64DFR2_EL1) || boolean IMPLEMENTATION_DEFINED "ID_AA64DFR2_EL1 trapped by
        ↪HCR_EL2.TID3") && HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ID_AA64DFR2_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ID_AA64DFR2_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.43 ID_AA64ISAR0_EL1, AArch64 Instruction Set Attribute Register 0

The ID_AA64ISAR0_EL1 characteristics are:

Purpose

Provides information about the instructions implemented in AArch64 state.

For general information about the interpretation of the ID registers, see 'Principles of the ID scheme for fields in ID registers'.

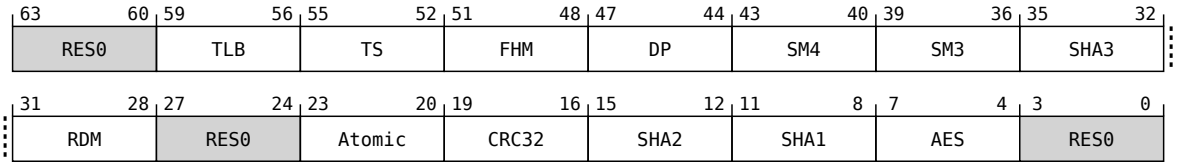
Configuration

There are no configuration notes.

Attributes

ID_AA64ISAR0_EL1 is a 64-bit register.

Field descriptions



Bits [63:60]

Reserved, RES0.

TLB, bits [59:56]

Indicates support for Outer Shareable and TLB range maintenance instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TLB	Meaning
0b0000	Outer Shareable and TLB range maintenance instructions are not implemented.
0b0001	Outer Shareable TLB maintenance instructions are implemented.
0b0010	Outer Shareable and TLB range maintenance instructions are implemented.

All other values are reserved.

FEAT_TLBIOS implements the functionality identified by the values 0b0001 and 0b0010.

FEAT_TLBIRANGE implements the functionality identified by the value 0b0010.

From Armv8.4, the values 0b0000 and 0b0001 are not permitted.

Access to this field is RO.

TS, bits [55:52]

Indicates support for flag manipulation instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

All other values are reserved.

FEAT_FlagM implements the functionality identified by the value 0b0001.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

In Armv8.4, the value 0b0000 is not permitted.

Access to this field is RO.

FHM, bits [51:48]

Indicates support for the Advanced SIMD half-precision to single-precision multiply instructions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

FHM	Meaning
0b0000	The Advanced SIMD half-precision to single-precision multiply instructions are not implemented.
0b0001	FMLAL and FMLSL instructions are implemented.

All other values are reserved.

FEAT_FHM implements the functionality identified by the value 0b0001.

From Armv8.4, if FEAT_FP16 is implemented, the value 0b0000 is not permitted.

Access to this field is RO.

DP, bits [47:44]

Indicates support for Dot Product instructions in AArch64 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

DP	Meaning
0b0000	No Dot Product instructions implemented.
0b0001	UDOT and SDOT instructions implemented.

All other values are reserved.

FEAT_DotProd implements the functionality identified by the value 0b0001.

From Armv8.4, if FEAT_AdvSIMD is implemented, the value 0b0000 is not permitted.

Access to this field is RO.

SM4, bits [43:40]

Indicates support for SM4 instructions in AArch64 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SM4	Meaning
0b0000	No SM4 instructions implemented.
0b0001	SM4E and SM4EKEY instructions implemented.

All other values are reserved.

If FEAT_SM4 is not implemented, the value 0b0001 is reserved.

FEAT_SM4 implements the functionality identified by the value 0b0001.

This field must have the same value as ID_AA64ISAR0_EL1.SM3.

Access to this field is RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SM3, bits [39:36]

Indicates support for the following SM3 instructions SM3SS1, SM3TT1A, SM3TT1B, SM3TT2A, SM3TT2B, SM3PARTW1, and SM3PARTW2 in AArch64 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SM3	Meaning
0b0000	The specified instructions are not implemented.
0b0001	The specified instructions are implemented.

All other values are reserved.

If FEAT_SM3 is not implemented, the value 0b0001 is reserved.

FEAT_SM3 implements the functionality identified by the value 0b0001.

This field must have the same value as ID_AA64ISAR0_EL1.SM4.

Access to this field is RO.

SHA3, bits [35:32]

Indicates support for the following SHA3 instructions EOR3, RAX1, XAR, and BCAX in AArch64 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SHA3	Meaning
0b0000	The specified instructions are not implemented.
0b0001	The specified instructions are implemented.

All other values are reserved.

If FEAT_SHA3 is not implemented, the value 0b0001 is reserved.

FEAT_SHA3 implements the functionality identified by the value 0b0001.

If the value of ID_AA64ISAR0_EL1.SHA1 is 0b0000, this field must have the value 0b0000.

If the value of this field is 0b0001, ID_AA64ISAR0_EL1.SHA2 must have the value 0b0010.

Access to this field is RO.

RDM, bits [31:28]

Indicates support for SQRDMLAH and SQRDMLSH instructions in AArch64 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

RDM	Meaning
0b0000	No RDMA instructions implemented.
0b0001	SQRDMLAH and SQRDMLSH instructions implemented.

All other values are reserved.

FEAT_RDM implements the functionality identified by the value 0b0001.

From Armv8.1, the value 0b0000 is not permitted.

Access to this field is RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [27:24]

Reserved, RES0.

Atomic, bits [23:20]

Indicates support for Atomic instructions in AArch64 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

All other values are reserved.

FEAT_LSE implements the functionality identified by the value 0b0010.

From Armv8.1, the value 0b0000 is not permitted.

Access to this field is RO.

CRC32, bits [19:16]

Indicates support for the following CRC32 instructions CRC32B, CRC32H, CRC32W, CRC32X, CRC32CB, CRC32CH, CRC32CW, and CRC32CX in AArch64 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CRC32	Meaning
0b0000	The specified instructions are not implemented.
0b0001	The specified instructions are implemented.

All other values are reserved.

FEAT_CRC32 implements the functionality identified by the value 0b0001.

From Armv8.1, the value 0b0000 is not permitted.

Access to this field is RO.

SHA2, bits [15:12]

Indicates support for SHA2 instructions in AArch64 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SHA2	Meaning
0b0000	No SHA2 instructions implemented.
0b0001	Implements instructions: SHA256H, SHA256H2, SHA256SU0, and SHA256SU1.
0b0010	Implements instructions: • SHA256H, SHA256H2, SHA256SU0, and SHA256SU1. • SHA512H, SHA512H2, SHA512SU0, and SHA512SU1.

All other values are reserved.

FEAT_SHA256 implements the functionality identified by the value 0b0001.

FEAT_SHA512 implements the functionality identified by the value 0b0010.

If the value of ID_AA64ISAR0_EL1.SHA1 is 0b0000, this field must have the value 0b0000.

If the value of this field is 0b0010, ID_AA64ISAR0_EL1.SHA3 must have the value 0b0001.

Access to this field is RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SHA1, bits [11:8]

Indicates support for the following SHA1 instructions SHA1C, SHA1P, SHA1M, SHA1H, SHA1SU0, and SHA1SU1 in AArch64 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SHA1	Meaning
0b0000	The specified instructions are not implemented.
0b0001	The specified instructions are implemented.

All other values are reserved.

FEAT_SHA1 implements the functionality identified by the value 0b0001.

If the value of ID_AA64ISAR0_EL1.SHA2 is 0b0000, this field must have the value 0b0000.

Access to this field is RO.

AES, bits [7:4]

Indicates support for AES instructions in AArch64 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

AES	Meaning
0b0000	No AES instructions implemented.
0b0001	AESE, AESD, AESMC, and AESIMC instructions implemented.
0b0010	As for 0b0001, plus PMULL and PMULL2 instructions operating on 64-bit source elements.

FEAT_AES implements the functionality identified by the value 0b0001.

FEAT_PMULL implements the functionality identified by the value 0b0010.

All other values are reserved.

Access to this field is RO.

Bits [3:0]

Reserved, RES0.

Accessing ID_AA64ISAR0_EL1

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, ID_AA64ISAR0_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0110	0b000

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ID_AA64ISAR0_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ID_AA64ISAR0_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.44 ID_AA64ISAR1_EL1, AArch64 Instruction Set Attribute Register 1

The ID_AA64ISAR1_EL1 characteristics are:

Purpose

Provides information about the features and instructions implemented in AArch64 state.

For general information about the interpretation of the ID registers, see 'Principles of the ID scheme for fields in ID registers'.

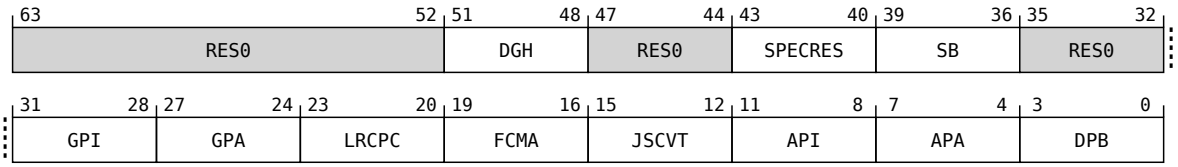
Configuration

There are no configuration notes.

Attributes

ID_AA64ISAR1_EL1 is a 64-bit register.

Field descriptions



Bits [63:52]

Reserved, RES0.

DGH, bits [51:48]

Indicates support for the Data Gathering Hint instruction.

The value of this field is an IMPLEMENTATION DEFINED choice of:

DGH	Meaning
0b0000	Data Gathering Hint is not implemented.
0b0001	Data Gathering Hint is implemented.

All other values are reserved.

FEAT_DGH implements the functionality identified by 0b0001.

From Armv8.0, the permitted values are 0b0000 and 0b0001.

If the DGH instruction has no effect in preventing the merging of memory accesses, the value of this field is 0b0000.

Access to this field is RO.

Bits [47:44]

Reserved, RES0.

SPECRES, bits [43:40]

Indicates support for prediction invalidation instructions in AArch64 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

All other values are reserved.

FEAT_SPECRES implements the functionality identified by 0b0001.

In Armv8-R, the permitted values are 0b0000 and 0b0001.

Access to this field is RO.

SB, bits [39:36]

Indicates support for SB instruction in AArch64 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SB	Meaning
0b0000	SB instruction is not implemented.
0b0001	SB instruction is implemented.

All other values are reserved.

FEAT_SB implements the functionality identified by 0b0001.

In Armv8-R, the permitted values are 0b0000 and 0b0001.

Access to this field is RO.

Bits [35:32]

Reserved, RES0.

GPI, bits [31:28]

Indicates support for an IMPLEMENTATION DEFINED algorithm is implemented in the PE for generic code authentication in AArch64 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

GPI	Meaning
0b0000	Generic Authentication using an IMPLEMENTATION DEFINED algorithm is not implemented.
0b0001	Generic Authentication using an IMPLEMENTATION DEFINED algorithm is implemented. This includes the PACGA instruction.

All other values are reserved.

When this field is nonzero, FEAT_PACIMP is implemented.

From Armv8.3, the permitted values are 0b0000 and 0b0001.

If the value of ID_AA64ISAR1_EL1.GPA is nonzero, this field must have the value 0b0000.

Access to this field is RO.

GPA, bits [27:24]

Indicates whether the QARMA5 algorithm is implemented in the PE for generic code authentication in AArch64 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

GPA	Meaning
0b0000	Generic Authentication using the QARMA5 algorithm is not implemented.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

GPA	Meaning
0b0001	Generic Authentication using the QARMA5 algorithm is implemented. This includes the PACGA instruction.

All other values are reserved.

FEAT_PACQARMA5 implements the functionality identified by 0b0001.

When this field is nonzero, FEAT_PACQARMA5 is implemented.

From Armv8.3, the permitted values are 0b0000 and 0b0001.

If the value of ID_AA64ISAR1_EL1.GPI is nonzero, this field must have the value 0b0000.

Access to this field is RO.

LRCPC, bits [23:20]

Indicates support for weaker release consistency, RCpc, based model.

The value of this field is an IMPLEMENTATION DEFINED choice of:

All other values are reserved.

FEAT_LRCPC implements the functionality identified by the value 0b0001.

FEAT_LRCPC2 implements the functionality identified by the value 0b0010.

From Armv8.3, the value 0b0000 is not permitted.

From Armv8.4, the value 0b0001 is not permitted.

Access to this field is RO.

FCMA, bits [19:16]

Indicates support for complex number addition and multiplication, where numbers are stored in vectors.

The value of this field is an IMPLEMENTATION DEFINED choice of:

FCMA	Meaning
0b0000	The FCMLA and FCADD instructions are not implemented.
0b0001	The FCMLA and FCADD instructions are implemented.

All other values are reserved.

FEAT_FCMA implements the functionality identified by the value 0b0001.

From Armv8.3, if Advanced SIMD or floating-point is implemented, the value 0b0000 is not permitted.

From Armv8.3, if Advanced SIMD or floating-point is not implemented, the only permitted value is 0b0000.

Access to this field is RO.

JSCVT, bits [15:12]

Indicates support for JavaScript conversion from double-precision floating-point values to integers in AArch64 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

JSCVT	Meaning
0b0000	The FJCVTZS instruction is not implemented.
0b0001	The FJCVTZS instruction is implemented.

All other values are reserved.

FEAT_JSCVT implements the functionality identified by 0b0001.

From Armv8.3, if Advanced SIMD or floating-point is implemented, the value 0b0000 is not permitted.

From Armv8.3, if Advanced SIMD or floating-point is not implemented, the only permitted value is 0b0000.

Access to this field is RO.

API, bits [11:8]

Indicates whether an IMPLEMENTATION DEFINED algorithm is implemented in the PE for address authentication, in AArch64 state. This applies to all Pointer Authentication instructions other than the PACGA instruction.

The value of this field is an IMPLEMENTATION DEFINED choice of:

All other values are reserved.

FEAT_PAuth implements the functionality identified by 0b0001.

FEAT_EPAC implements the functionality identified by 0b0010.

FEAT_PAuth2 implements the functionality identified by 0b0011.

FEAT_FPAC implements the functionality identified by 0b0100.

FEAT_FPACCOMBINE implements the functionality identified by 0b0101.

When this field is nonzero, FEAT_PACIMP is implemented.

In Armv8-R, the permitted values are 0b0000, 0b0001, 0b0010, 0b0011, 0b0100, and 0b0101.

If the value of ID_AA64ISAR1_EL1.APA is nonzero, this field must have the value 0b0000.

Access to this field is RO.

APA, bits [7:4]

Indicates whether the QARMA5 algorithm is implemented in the PE for address authentication, in AArch64 state. This applies to all Pointer Authentication instructions other than the PACGA instruction.

The value of this field is an IMPLEMENTATION DEFINED choice of:

All other values are reserved.

FEAT_PAuth implements the functionality identified by 0b0001.

FEAT_EPAC implements the functionality identified by 0b0010.

FEAT_PAuth2 implements the functionality identified by 0b0011.

FEAT_FPAC implements the functionality identified by 0b0100.

FEAT_FPACCOMBINE implements the functionality identified by 0b0101.

When this field is nonzero, FEAT_PACQARMA5 is implemented.

In Armv8-R, the permitted values are 0b0000, 0b0001, 0b0010, 0b0011, 0b0100, and 0b0101.

If the value of ID_AA64ISAR1_EL1.API is nonzero, this field must have the value 0b0000.

Access to this field is RO.

DPB, bits [3:0]

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Data Persistence writeback. Indicates support for the DC CVAP and DC CVADP instructions in AArch64 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

DPB	Meaning
0b0000	DC CVAP not supported.
0b0001	DC CVAP supported.
0b0010	DC CVAP and DC CVADP supported.

All other values are reserved.

FEAT_DPB implements the functionality identified by the value 0b0001.

FEAT_DPB2 implements the functionality identified by the value 0b0010.

In Armv8-R, the value 0b0000 is not permitted.

Access to this field is RO.

Accessing ID_AA64ISAR1_EL1

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, ID_AA64ISAR1_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0110	0b001

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ID_AA64ISAR1_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ID_AA64ISAR1_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.45 ID_AA64ISAR2_EL1, AArch64 Instruction Set Attribute Register 2

The ID_AA64ISAR2_EL1 characteristics are:

Purpose

Provides information about the features and instructions implemented in AArch64 state.

For general information about the interpretation of the ID registers, see 'Principles of the ID scheme for fields in ID registers'.

Configuration

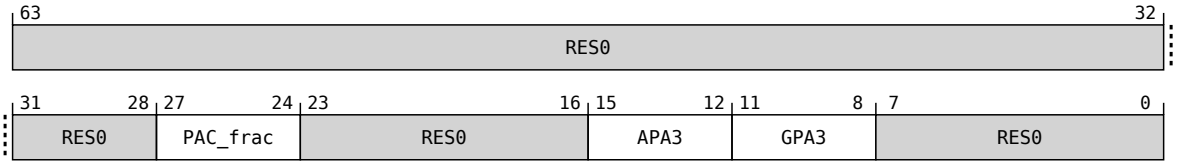
Note

Prior to the introduction of the features described by this register, this register was unnamed and reserved, RES0 from EL1 and EL2.

Attributes

ID_AA64ISAR2_EL1 is a 64-bit register.

Field descriptions



Bits [63:28]

Reserved, RES0.

PAC_frac, bits [27:24]

Indicates which address bit is used to determine the size of the PAC field.

The value of this field is an IMPLEMENTATION DEFINED choice of:

PAC_frac	Meaning
0b0000	The address bit which is used to define the size of the PAC field is dependent on whether address tagging is used.
0b0001	The address bit which is used to define the size of the PAC field is fixed.

All other values are reserved.

FEAT_CONSTPACFIELD implements the functionality identified by the value 0b0001.

From Armv8.3, the permitted values are 0b0000 and 0b0001.

Access to this field is RO.

Bits [23:16]

Reserved, RES0.

APA3, bits [15:12]

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Indicates whether the QARMA3 algorithm is implemented in the PE for address authentication in AArch64 state. This applies to all Pointer Authentication instructions other than the PACGA instruction.

The value of this field is an IMPLEMENTATION DEFINED choice of:

All other values are reserved.

FEAT_PAuth implements the functionality identified by the value 0b0001.

FEAT_EPAC implements the functionality identified by the value 0b0010.

FEAT_PAuth2 implements the functionality identified by the value 0b0011.

FEAT_FPAC implements the functionality identified by the value 0b0100.

FEAT_FPACCOMBINE implements the functionality identified by the value 0b0101.

When this field is nonzero, FEAT_PACQARMA3 is implemented.

In Armv8.3, the permitted values are 0b0000, 0b0001, 0b0010, 0b0011, 0b0100, and 0b0101.

From Armv8.6, the permitted values are 0b0000, 0b0011, 0b0100, and 0b0101.

If the value of ID_AA64ISAR1_EL1.API is nonzero, or the value of ID_AA64ISAR1_EL1.APA is nonzero, this field must have the value 0b0000.

Access to this field is RO.

GPA3, bits [11:8]

Indicates whether the QARMA3 algorithm is implemented in the PE for generic code authentication in AArch64 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

GPA3	Meaning
0b0000	Generic Authentication using the QARMA3 algorithm is not implemented.
0b0001	Generic Authentication using the QARMA3 algorithm is implemented. This includes the PACGA instruction.

All other values are reserved.

FEAT_PACQARMA3 implements the functionality identified by the value 0b0001.

When this field is nonzero, FEAT_PACQARMA3 is implemented.

From Armv8.3, the permitted values are 0b0000 and 0b0001.

If the value of ID_AA64ISAR1_EL1.GPI is nonzero, or the value of ID_AA64ISAR1_EL1.GPA is nonzero, this field must have the value 0b0000.

Access to this field is RO.

Bits [7:0]

Reserved, RES0.

Accessing ID_AA64ISAR2_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ID_AA64ISAR2_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0110	0b010

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if (!IsZero(ID_AA64ISAR2_EL1) || boolean IMPLEMENTATION_DEFINED "ID_AA64ISAR2_EL1 trapped by
        ↪HCR_EL2.TID3") && HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ID_AA64ISAR2_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ID_AA64ISAR2_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.46 ID_AA64MMFR0_EL1, AArch64 Memory Model Feature Register 0

The ID_AA64MMFR0_EL1 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch64 state.

For general information about the interpretation of the ID registers, see 'Principles of the ID scheme for fields in ID registers'.

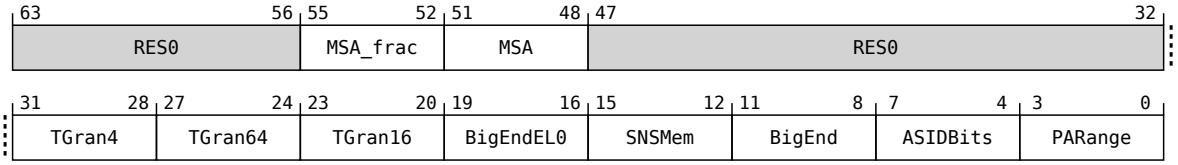
Configuration

There are no configuration notes.

Attributes

ID_AA64MMFR0_EL1 is a 64-bit register.

Field descriptions



Bits [63:56]

Reserved, RES0.

MSA_frac, bits [55:52]

Memory System Architecture fractional field. This holds the information on additional Memory System Architectures supported.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MSA_frac	Meaning
0b0000	PMSAv8-64 not supported in any translation regime.
0b0001	PMSAv8-64 supported in all translation regimes. No support for VMSAv8-64.
0b0010	PMSAv8-64 supported in all translation regimes. In addition to PMSAv8-64, stage 1 EL1&0 translation regime also supports VMSAv8-64.

All other values are reserved.

The permitted values are 0b0001 and 0b0010.

This field is valid only when ID_AA64MMFR0_EL1.MSA is 0b1111.

Access to this field is RO.

MSA, bits [51:48]

Memory System Architecture ID field. This holds the information on Memory System Architectures supported.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MSA	Meaning
0b0000	VMSAv8-64 supported in all translation regimes. No support for PMSAv8-64.
0b1111	See ID_AA64MMFR0_EL1.MSA_frac for the Memory System Architectures supported.

All other values are reserved.

In Armv8-R, the only permitted value is 0b1111.

Access to this field is RO.

Bits [47:32]

Reserved, RES0.

TGran4, bits [31:28]

Indicates support for 4KB memory translation granule size.

The value of this field is an IMPLEMENTATION DEFINED choice of:

All other values are reserved.

Access to this field is RO.

TGran64, bits [27:24]

Indicates support for 64KB memory translation granule size.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TGran64	Meaning
0b0000	64KB granule supported.
0b1111	64KB granule not supported.

All other values are reserved.

Access to this field is RO.

TGran16, bits [23:20]

Indicates support for 16KB memory translation granule size.

The value of this field is an IMPLEMENTATION DEFINED choice of:

All other values are reserved.

Access to this field is RO.

BigEndEL0, bits [19:16]

Indicates support for mixed-endian at EL0 only.

The value of this field is an IMPLEMENTATION DEFINED choice of:

BigEndEL0	Meaning
0b0000	No mixed-endian support at EL0. The SCTLR_EL1.E0E bit has a fixed value.
0b0001	Mixed-endian support at EL0. The SCTLR_EL1.E0E bit can be configured.

FEAT_MixedEndEL0 implements the functionality identified by the value 0b0001.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

All other values are reserved.

This field is invalid and is RES0 if ID_AA64MMFR0_EL1.BigEnd is not 0b0000.

Access to this field is RO.

SNSMem, bits [15:12]

Indicates support for a distinction between Secure and Non-secure Memory.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SNSMem	Meaning
0b0000	Does not support a distinction between Secure and Non-secure Memory.
0b0001	Does support a distinction between Secure and Non-secure Memory.

Note

If EL3 is implemented, the value 0b0000 is not permitted.

All other values are reserved.

Access to this field is RO.

BigEnd, bits [11:8]

Indicates support for mixed-endian configuration.

The value of this field is an IMPLEMENTATION DEFINED choice of:

BigEnd	Meaning
0b0000	No mixed-endian support. The SCTLR_ELx.EE bits have a fixed value. See the BigEndEL0 field, bits[19:16], for whether EL0 supports mixed-endian.
0b0001	Mixed-endian support. The SCTLR_ELx.EE and SCTLR_EL1.E0E bits can be configured.

FEAT_MixedEnd implements the functionality identified by the value 0b0001.

If this field is 0b0001, FEAT_MixedEndEL0 is also implemented.

All other values are reserved.

Access to this field is RO.

ASIDBits, bits [7:4]

Number of ASID bits.

The value of this field is an IMPLEMENTATION DEFINED choice of:

ASIDBits	Meaning
0b0000	8 bits.
0b0010	16 bits.

All other values are reserved.

Access to this field is RO.

PARange, bits [3:0]

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Physical Address range supported.

The value of this field is an IMPLEMENTATION DEFINED choice of:

PARange	Meaning	Applies when
0b0000	32 bits, 4GB.	
0b0001	36 bits, 64GB.	
0b0010	40 bits, 1TB.	
0b0011	42 bits, 4TB.	
0b0100	44 bits, 16TB.	
0b0101	48 bits, 256TB.	
0b0110	52 bits, 4PB.	FEAT_LPA is implemented

All other values are reserved.

Access to this field is RO.

Accessing ID_AA64MMFR0_EL1

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, ID_AA64MMFR0_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0111	0b000

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ID_AA64MMFR0_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ID_AA64MMFR0_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.47 ID_AA64MMFR1_EL1, AArch64 Memory Model Feature Register 1

The ID_AA64MMFR1_EL1 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch64 state.

For general information about the interpretation of the ID registers, see 'Principles of the ID scheme for fields in ID registers'.

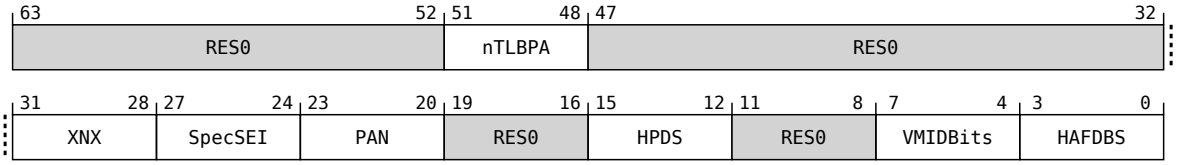
Configuration

There are no configuration notes.

Attributes

ID_AA64MMFR1_EL1 is a 64-bit register.

Field descriptions



Bits [63:52]

Reserved, RES0.

nTLBPA, bits [51:48]

When architecture implements VMSA extension:

Indicates support for intermediate caching of translation table walks.

The value of this field is an IMPLEMENTATION DEFINED choice of:

nTLBPA	Meaning
0b0000	The intermediate caching of translation table walks might include non-coherent physical translation caches.
0b0001	The intermediate caching of translation table walks does not include non-coherent physical translation caches.

Non-coherent physical translation caches are non-coherent caches of previous valid translation table entries since the last completed relevant TLBI applicable to the PE, where either:

- The caching is indexed by the physical address of the location holding the translation table entry.
- The caching is used for stage 1 translations and is indexed by the intermediate physical address of the location holding the translation table entry.

All other values are reserved.

FEAT_nTLBPA implements the functionality identified by the value 0b0001.

Access to this field is RO.

Otherwise:

Reserved, RES0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [47:32]

Reserved, RES0.

XNX, bits [31:28]

Indicates support for execute-never control distinction by Exception level at stage 2.

The value of this field is an IMPLEMENTATION DEFINED choice of:

XNX	Meaning
0b0000	Distinction between EL0 and EL1 execute-never control at stage 2 not supported.
0b0001	Distinction between EL0 and EL1 execute-never control at stage 2 supported.

All other values are reserved.

FEAT_XNX implements the functionality identified by the value 0b0001.

From Armv8.2, the value 0b0000 is not permitted.

Access to this field is RO.

SpecSEI, bits [27:24]

When FEAT_RAS is implemented:

Describes whether the PE can generate SError exceptions from speculative reads of memory, including speculative instruction fetches.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SpecSEI	Meaning
0b0000	The PE never generates an SError exception due to an External abort on a speculative read.
0b0001	The PE might generate an SError exception due to an External abort on a speculative read.

All other values are reserved.

FEAT_SpecSEI implements the functionality identified by the value 0b0001.

Access to this field is RO.

Otherwise:

Reserved, RES0.

PAN, bits [23:20]

Privileged Access Never. Indicates support for the PAN bit in PSTATE, SPSR_EL1, SPSR_EL2, SPSR_EL3, and DSPSR_EL0.

The value of this field is an IMPLEMENTATION DEFINED choice of:

All other values are reserved.

FEAT_PAN implements the functionality identified by the value 0b0001.

FEAT_PAN2 implements the functionality added by the value 0b0010.

From Armv8.1, the value 0b0000 is not permitted.

From Armv8.2, the value 0b0001 is not permitted.

Access to this field is RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [19:16]

Reserved, RES0.

HPDS, bits [15:12]

When architecture implements VMSA extension:

Hierarchical Permission Disables. Indicates support for disabling hierarchical controls in translation tables.

The value of this field is an IMPLEMENTATION DEFINED choice of:

HPDS	Meaning
0b0000	Disabling of hierarchical controls not supported.
0b0001	Disabling of hierarchical controls supported with the TCR_EL1.{HPD1, HPD0}, TCR_EL2.HPD or TCR_EL2.{HPD1, HPD0}, and TCR_EL3.HPD bits.
0b0010	As for value 0b0001, and adds possible hardware allocation of bits[62:59] of the Translation table descriptors from the final lookup level for IMPLEMENTATION DEFINED use.

All other values are reserved.

FEAT_HPDS implements the functionality identified by the value 0b0001.

FEAT_HPDS2 implements the functionality identified by the value 0b0010.

From Armv8.1, the value 0b0000 is not permitted.

Access to this field is RO.

Otherwise:

Reserved, RES0.

Bits [11:8]

Reserved, RES0.

VMIDBits, bits [7:4]

Number of VMID bits.

The value of this field is an IMPLEMENTATION DEFINED choice of:

VMIDBits	Meaning
0b0000	8 bits
0b0010	16 bits

All other values are reserved.

FEAT_VMID16 implements the functionality identified by the value 0b0010.

Access to this field is RO.

HAFDBS, bits [3:0]

When architecture implements VMSA extension:

Hardware updates to Access flag and Dirty state in translation tables.

The value of this field is an IMPLEMENTATION DEFINED choice of:

All other values are reserved.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_HAFDBS implements the functionality identified by the values 0b0001 and 0b0010.

Access to this field is RO.

Otherwise:

Reserved, RES0.

Accessing ID_AA64MMFR1_EL1

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, ID_AA64MMFR1_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0111	0b001

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ID_AA64MMFR1_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ID_AA64MMFR1_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.48 ID_AA64MMFR2_EL1, AArch64 Memory Model Feature Register 2

The ID_AA64MMFR2_EL1 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch64 state.

For general information about the interpretation of the ID registers, see 'Principles of the ID scheme for fields in ID registers'.

Configuration

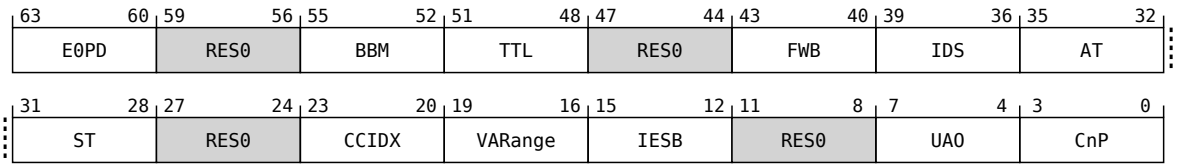
Note

Prior to the introduction of the features described by this register, this register was unnamed and reserved, RES0 from EL1 and EL2.

Attributes

ID_AA64MMFR2_EL1 is a 64-bit register.

Field descriptions



E0PD, bits [63:60]

When architecture implements VMSA extension:

Indicates support for the E0PD mechanism.

The value of this field is an IMPLEMENTATION DEFINED choice of:

E0PD	Meaning
0b0000	E0PDx mechanism is not implemented.
0b0001	E0PDx mechanism is implemented.

All other values are reserved.

FEAT_E0PD implements the functionality identified by the value 0b0001.

In Armv8.4, the permitted values are 0b0000 and 0b0001.

If FEAT_E0PD is implemented, FEAT_CSV3 must be implemented.

Access to this field is RO.

Otherwise:

Reserved, RES0.

Bits [59:56]

Reserved, RES0.

BBM, bits [55:52]

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When architecture implements VMSA extension:

Allows identification of the requirements of the hardware to have break-before-make sequences when changing block or table size for a translation.

The value of this field is an IMPLEMENTATION DEFINED choice of:

All other values are reserved.

FEAT_BBML1 implements the functionality identified by the value 0b0001.

FEAT_BBML2 implements the functionality identified by the value 0b0010.

Access to this field is RO.

Otherwise:

Reserved, RES0.

TTL, bits [51:48]

When architecture implements VMSA extension:

Indicates support for TTL field in address operations.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TTL	Meaning
0b0000	TLB maintenance instructions by address have bits[47:44] as RES0.
0b0001	TLB maintenance instructions by address have bits[47:44] holding the TTL field.

All other values are reserved.

FEAT_TTL implements the functionality identified by the value 0b0001.

This field affects TLBI IPAS2E1, TLBI IPAS2E1IS, TLBI IPAS2E1OS, TLBI IPAS2LE1, TLBI IPAS2LE1IS, TLBI IPAS2LE1OS, TLBI VAAE1, TLBI VAAE1IS, TLBI VAAE1OS, TLBI VAALE1, TLBI VAALE1IS, TLBI VAALE1OS, TLBI VAE1, TLBI VAE1IS, TLBI VAE1OS, TLBI VAE2, TLBI VAE2IS, TLBI VAE2OS, TLBI VAE3, TLBI VAE3IS, TLBI VAE3OS,TLBI VALE1, TLBI VALE1IS, TLBI VALE1OS, TLBI VALE2, TLBI VALE2IS, TLBI VALE2OS, TLBI VALE3, TLBI VALE3IS, TLBI VALE3OS.

From Armv8.4, the value 0b0000 is not permitted.

Access to this field is RO.

Otherwise:

Reserved, RES0.

Bits [47:44]

Reserved, RES0.

FWB, bits [43:40]

Indicates support for HCR_EL2.FWB.

The value of this field is an IMPLEMENTATION DEFINED choice of:

FWB	Meaning
0b0000	HCR_EL2.FWB bit is not supported.
0b0001	HCR_EL2.FWB is supported.

All other values reserved.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_S2FWB implements the functionality identified by the value 0b0001.

From Armv8.4, the value 0b0000 is not permitted.

Access to this field is RO.

IDS, bits [39:36]

Indicates the EC syndrome value in ESR_ELx.EC that is reported if an exception is generated by a read access to the feature ID space.

The value of this field is an IMPLEMENTATION DEFINED choice of:

All other values are reserved.

The Feature ID space is defined as the System register space in AArch64 with op0==3, op1=={0, 1, 3}, CRn==0, CRm=={0-7}, op2=={0-7}.

FEAT_IDST implements the functionality identified by the value 0b0001.

From Armv8.4, the value 0b0000 is not permitted.

Access to this field is RO.

AT, bits [35:32]

Identifies support for unaligned single-copy atomicity and atomic functions.

The value of this field is an IMPLEMENTATION DEFINED choice of:

AT	Meaning
0b0000	Unaligned single-copy atomicity and atomic functions are not supported.
0b0001	Unaligned single-copy atomicity and atomic functions with a 16-byte address range aligned to 16-bytes are supported.

All other values are reserved.

FEAT_LSE2 implements the functionality identified by the value 0b0001.

From Armv8.4, the value 0b0000 is not permitted.

Access to this field is RO.

ST, bits [31:28]

When architecture implements VMSA extension:

Identifies support for small translation tables.

The value of this field is an IMPLEMENTATION DEFINED choice of:

ST	Meaning
0b0000	The maximum value of the TCR_ELx.{T0SZ,T1SZ} and VTCR_EL2.T0SZ fields is 39.
0b0001	The maximum value of the TCR_ELx.{T0SZ,T1SZ} and VTCR_EL2.T0SZ fields is 48 for 4KB and 16KB granules, and 47 for 64KB granules.

All other values are reserved.

FEAT_TTST implements the functionality identified by the value 0b0001.

When FEAT_SEL2 is implemented, the value 0b0000 is not permitted.

Access to this field is RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bits [27:24]

Reserved, RES0.

CCIDX, bits [23:20]

Support for the use of revised CCSIDR_EL1 register format.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CCIDX	Meaning
0b0000	32-bit format implemented for all levels of the CCSIDR_EL1.
0b0001	64-bit format implemented for all levels of the CCSIDR_EL1.

All other values are reserved.

FEAT_CCIDX implements the functionality identified by the value 0b0001.

From Armv8.3, the permitted values are 0b0000 and 0b0001.

Access to this field is RO.

VARange, bits [19:16]

Indicates support for a larger virtual address.

The value of this field is an IMPLEMENTATION DEFINED choice of:

All other values are reserved.

FEAT_LVA implements the functionality identified by the value 0b0001.

Access to this field is RO.

IESB, bits [15:12]

Indicates support for the IESB bit in the SCTLR_ELx registers.

The value of this field is an IMPLEMENTATION DEFINED choice of:

IESB	Meaning
0b0000	IESB bit in the SCTLR_ELx registers is not supported.
0b0001	IESB bit in the SCTLR_ELx registers is supported.

All other values are reserved.

FEAT_IESB implements the functionality identified by the value 0b0001.

Access to this field is RO.

Bits [11:8]

Reserved, RES0.

UAO, bits [7:4]

User Access Override.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

UAO	Meaning
0b0000	UAO not supported.
0b0001	UAO supported.

All other values are reserved.

FEAT_UAO implements the functionality identified by the value 0b0001.

From Armv8.2, the value 0b0000 is not permitted.

Access to this field is RO.

CnP, bits [3:0]

When architecture implements VMSA extension:

Indicates support for Common not Private translations.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CnP	Meaning
0b0000	Common not Private translations not supported.
0b0001	Common not Private translations supported.

All other values are reserved.

FEAT_TTCNP implements the functionality identified by the value 0b0001.

From Armv8.2, the value 0b0000 is not permitted.

Access to this field is RO.

Otherwise:

Reserved, RES0.

Accessing ID_AA64MMFR2_EL1

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, ID_AA64MMFR2_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0111	0b010

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if (!IsZero(ID_AA64MMFR2_EL1) || boolean IMPLEMENTATION_DEFINED "ID_AA64MMFR2_EL1 trapped by
        ↪HCR_EL2.TID3") && HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ID_AA64MMFR2_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ID_AA64MMFR2_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.49 ID_AA64MMFR3_EL1, AArch64 Memory Model Feature Register 3

The ID_AA64MMFR3_EL1 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch64 state.

Configuration

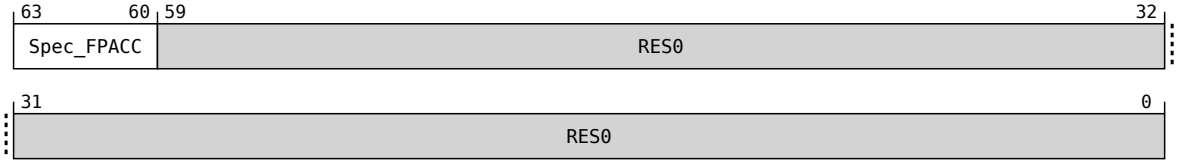
Note

Prior to the introduction of the features described by this register, this register was unnamed and reserved, RES0 from EL1 and EL2.

Attributes

ID_AA64MMFR3_EL1 is a 64-bit register.

Field descriptions



Spec_FPACC, bits [63:60]

When FEAT_FPACCOMBINE is implemented:

Speculative behavior in the event of a PAC authentication failure in an implementation that includes FEAT_FPACCOMBINE.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Spec_FPACC	Meaning
0b0000	The implementation does not disclose whether the speculative use of pointers processed by a PAC Authentication is materially different in terms of the impact on cached microarchitectural state between passing and failing of the PAC Authentication.
0b0001	The speculative use of pointers processed by a PAC Authentication is not materially different in terms of the impact on cached microarchitectural state between passing and failing of the PAC Authentication.

All other values are reserved.

For the purpose of this definition, cached microarchitecture state is the state of caching agents such as instruction caches, data caches and TLBs which can be altered as a result of speculation caused by a mispredicted execution, but is not restored to the state prior to the speculation when the misprediction is corrected.

Access to this field is RO.

Otherwise:

Reserved, RES0.

Bits [59:0]

Reserved, RES0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing ID_AA64MMFR3_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, ID_AA64MMFR3_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0111	0b011

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if (!IsZero(ID_AA64MMFR3_EL1) || boolean IMPLEMENTATION_DEFINED "ID_AA64MMFR3_EL1 trapped by
        ↪HCR_EL2.TID3") && HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ID_AA64MMFR3_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ID_AA64MMFR3_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.50 ID_AA64MMFR4_EL1, AArch64 Memory Model Feature Register 4

The ID_AA64MMFR4_EL1 characteristics are:

Purpose

Provides additional information about implemented memory model and memory management support in AArch64.

Configuration

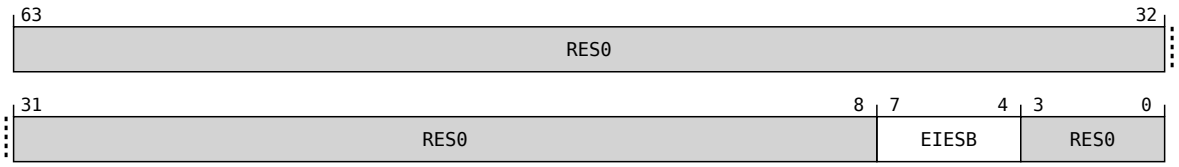
Note

Prior to the introduction of the features described by this register, this register was unnamed and reserved, RES0 from EL1 and EL2.

Attributes

ID_AA64MMFR4_EL1 is a 64-bit register.

Field descriptions



Bits [63:8]

Reserved, RES0.

EIESB, bits [7:4]

When FEAT_IESB is implemented:

Early Implicit Error Synchronization event. Indicates whether the implicit Error synchronization event inserted on taking an exception to ELx when SCTLR_ELx.IESB is 1 is inserted before or after the exception is taken.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EIESB	Meaning
0b1111	FEAT_IESB is implemented. When SError exceptions are routed to EL3, and either FEAT_DoubleFault is not implemented or the Effective value of SCR_EL3.NMEA is 1, an implicit Error synchronization event might be inserted after an exception is taken to EL3.
0b0000	FEAT_IESB is not implemented, or behavior is not described.
0b0001	FEAT_IESB is implemented. When SError exceptions are routed to EL3, and either FEAT_DoubleFault is not implemented or the Effective value of SCR_EL3.NMEA is 1, an implicit Error synchronization event is inserted before an exception taken to EL3.
0b0010	As 0b0001, and also: • When SError exceptions are routed to EL1, and either FEAT_DoubleFault2 is not implemented or the Effective value of SCTLR2_EL1.NMEA is 1, an implicit Error synchronization event is inserted before an exception taken to EL1. • When SError exceptions are routed to EL2, and either FEAT_DoubleFault2 is not implemented or the Effective value of SCTLR2_EL2.NMEA is 1, an implicit Error synchronization event is inserted before an exception taken to EL2.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

All other values are reserved.

This field describes the PE behavior on taking an exception to ELx when SCTLR_ELx.IESB is 1. This field does not describe the behavior when SCTLR_ELx.IESB is 0.

The behavior described by this field only applies for the conditions described above. For example, if ID_AA64MMFR4_EL1.EIESB reads as `0b0001`, then it does not describe the behavior when SError exceptions are not routed to EL3, or when FEAT_DoubleFault is implemented and the Effective value of SCR_EL3.NMEA is 0.

Inserting the event before the exception is taken means that if the Error synchronization event causes an SError exception to become pending, and SError exceptions are not masked and not disabled, then the SError exception is taken in place of the original exception.

When FEAT_IESB is not implemented, the only permitted value of this field is `0b0000`.

Access to this field is RO.

Otherwise:

Reserved, RES0.

Bits [3:0]

Reserved, RES0.

Accessing ID_AA64MMFR4_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, ID_AA64MMFR4_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0111	0b100

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if (!IsZero(ID_AA64MMFR4_EL1) || boolean IMPLEMENTATION_DEFINED "ID_AA64MMFR4_EL1 trapped by
        ↪HCR_EL2.TID3") && HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ID_AA64MMFR4_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ID_AA64MMFR4_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.51 ID_AA64PFR0_EL1, AArch64 Processor Feature Register 0

The ID_AA64PFR0_EL1 characteristics are:

Purpose

Provides additional information about implemented PE features in AArch64 state.

For general information about the interpretation of the ID registers, see 'Principles of the ID scheme for fields in ID registers'.

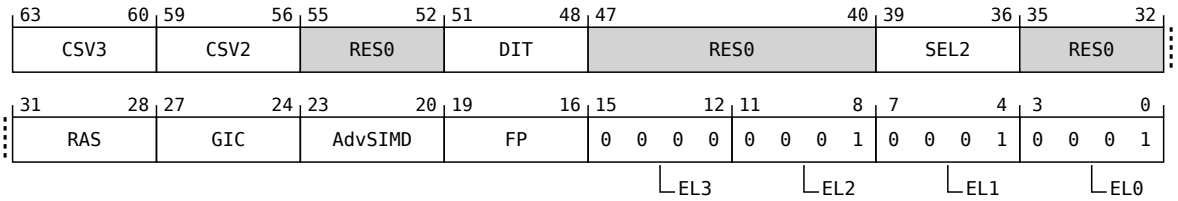
Configuration

The external register EDPFR gives information from this register.

Attributes

ID_AA64PFR0_EL1 is a 64-bit register.

Field descriptions



CSV3, bits [63:60]

Speculative use of faulting data.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CSV3	Meaning
0b0000	This PE does not disclose whether data loaded or read from a register under speculation where the data load or register read would not be permitted architecturally, can be used by instructions newer than the load or register read in a manner that allows the value of the inaccessible data to be recovered by code architecturally executed.
0b0001	Data loaded or read from a register under speculation where the data load or register read would not be permitted architecturally, cannot be used by instructions newer than the load or register read in a manner that allows the value of the inaccessible data to be recovered by code architecturally executed.

All other values are reserved.

FEAT_CSV3 implements the functionality identified by the value 0b0001.

If FEAT_E0PD is implemented, FEAT_CSV3 must be implemented.

From Armv8.5, the value 0b0000 is not permitted.

Access to this field is RO.

CSV2, bits [59:56]

Speculative use of out of context prediction resources.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CSV2	Meaning
0b0000	The implementation does not disclose whether FEAT_CSV2 is implemented.
0b0001	FEAT_CSV2 is implemented, but FEAT_CSV2_2 and FEAT_CSV2_3 are not implemented. ID_AA64PFR1_EL1.CSV2_frac determines whether either or both of FEAT_CSV2_1p1 or FEAT_CSV2_1p2 are implemented.
0b0010	FEAT_CSV2_2 is implemented, but FEAT_CSV2_3 is not implemented.
0b0011	FEAT_CSV2_3 is implemented.

All other values are reserved.

FEAT_CSV2 implements the functionality identified by the value 0b0001.

FEAT_CSV2_2 implements the functionality identified by the value 0b0010.

FEAT_CSV2_3 implements the functionality identified by the feature 0b0011.

Access to this field is RO.

Bits [55:52]

Reserved, RES0.

DIT, bits [51:48]

Data Independent Timing.

The value of this field is an IMPLEMENTATION DEFINED choice of:

All other values are reserved.

FEAT_DIT implements the functionality identified by the value 0b0001.

From Armv8.4, the value 0b0000 is not permitted.

Access to this field is RO.

Bits [47:40]

Reserved, RES0.

SEL2, bits [39:36]

Secure EL2.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SEL2	Meaning
0b0000	Secure EL2 is not implemented.
0b0001	Secure EL2 is implemented.

All other values are reserved.

FEAT_SEL2 implements the functionality identified by the value 0b0001.

From Armv8.4, if Secure state and EL2 are implemented, the value 0b0000 is not permitted.

From Armv8.4, if Secure state is not implemented, or if EL2 is not implemented, the only permitted value is 0b0000.

Access to this field is RO.

Bits [35:32]

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Reserved, RES0.

RAS, bits [31:28]

RAS Extension version.

The value of this field is an IMPLEMENTATION DEFINED choice of:

All other values are reserved.

FEAT_RAS implements the functionality identified by the value 0b0001.

FEAT_RASv1p1 and FEAT_DoubleFault implement the functionality identified by the value 0b0010.

In Armv8.0 and Armv8.1, the permitted values are 0b0000 and 0b0001.

From Armv8.2, the value 0b0000 is not permitted.

From Armv8.4, if FEAT_DoubleFault is implemented or ERRIDR_EL1.NUM is nonzero, the value 0b0001 is not permitted.

Note

When the value of this field is 0b0001, ID_AA64PFR1_EL1.RAS_frac indicates whether FEAT_RASv1p1 is implemented.

Access to this field is RO.

GIC, bits [27:24]

System register GIC CPU interface.

The value of this field is an IMPLEMENTATION DEFINED choice of:

GIC	Meaning
0b0000	GIC CPU interface system registers not implemented.
0b0001	System register interface to versions 3.0 and 4.0 of the GIC CPU interface is supported.
0b0011	System register interface to version 4.1 of the GIC CPU interface is supported.

All other values are reserved.

Access to this field is RO.

AdvSIMD, bits [23:20]

Advanced SIMD.

The value of this field is an IMPLEMENTATION DEFINED choice of:

AdvSIMD	Meaning
0b0000	Advanced SIMD is implemented, including support for the following SISD and SIMD operations: • Integer byte, halfword, word and doubleword element operations. • Single-precision and double-precision floating-point arithmetic. • Conversions between single-precision and half-precision data types, and double-precision and half-precision data types.
0b0001	As for 0b0000, and also includes support for half-precision floating-point arithmetic.
0b1111	Advanced SIMD is not implemented.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

All other values are reserved.

This field must have the same value as the FP field.

The permitted values are:

- 0b0000 in an implementation with Advanced SIMD support that does not include the FEAT_FP16 extension.
- 0b0001 in an implementation with Advanced SIMD support that includes the FEAT_FP16 extension.
- 0b1111 in an implementation without Advanced SIMD support.

Access to this field is RO.

FP, bits [19:16]

Floating-point.

The value of this field is an IMPLEMENTATION DEFINED choice of:

FP	Meaning
0b0000	Floating-point is implemented, and includes support for: • Single-precision and double-precision floating-point types. • Conversions between single-precision and half-precision data types, and double-precision and half-precision data types.
0b0001	As for 0b0000, and also includes support for half-precision floating-point arithmetic.
0b1111	Floating-point is not implemented.

All other values are reserved.

This field must have the same value as the AdvSIMD field.

The permitted values are:

- 0b0000 in an implementation with floating-point support that does not include the FEAT_FP16 extension.
- 0b0001 in an implementation with floating-point support that includes the FEAT_FP16 extension.
- 0b1111 in an implementation without floating-point support.

Access to this field is RO.

EL3, bits [15:12]

EL3 Exception level handling. In Armv8-R, the only permitted value is 0b0000. All other values are reserved.

EL3	Meaning
0b0000	EL3 is not implemented.

Access to this field is RO.

EL2, bits [11:8]

EL2 Exception level handling. In Armv8-R, the only permitted value is 0b0001. All other values are reserved.

EL2	Meaning
0b0001	EL2 can be executed in AArch64 state only.

Access to this field is RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EL1, bits [7:4]

EL1 Exception level handling. In Armv8-R, the only permitted value is `0b0001`. All other values are reserved.

EL1	Meaning
0b0001	EL1 can be executed in AArch64 state only.

Access to this field is RO.

EL0, bits [3:0]

EL0 Exception level handling. In Armv8-R, the only permitted value is `0b0001`. All other values are reserved.

EL0	Meaning
0b0001	EL0 can be executed in AArch64 state only.

Access to this field is RO.

Accessing ID_AA64PFR0_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, ID_AA64PFR0_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0100	0b000

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ID_AA64PFR0_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ID_AA64PFR0_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.52 ID_AA64PFR1_EL1, AArch64 Processor Feature Register 1

The ID_AA64PFR1_EL1 characteristics are:

Purpose

Provides additional information about implemented PE features in AArch64 state.

For general information about the interpretation of the ID registers, see 'Principles of the ID scheme for fields in ID registers'.

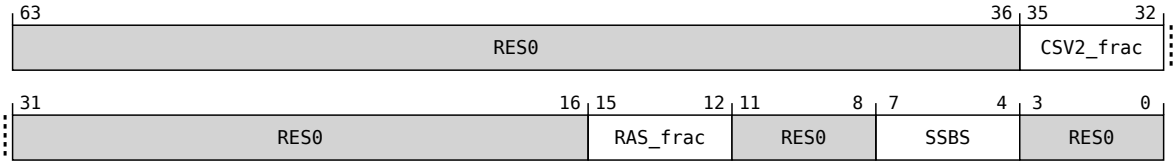
Configuration

There are no configuration notes.

Attributes

ID_AA64PFR1_EL1 is a 64-bit register.

Field descriptions



Bits [63:36]

Reserved, RES0.

CSV2_frac, bits [35:32]

CSV2 fractional field.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CSV2_frac	Meaning
0b0000	Either ID_AA64PFR0_EL1.CSV2 is not 0b0001, or the implementation does not disclose whether FEAT_CSV2_1p1 is implemented. FEAT_CSV2_1p2 is not implemented.
0b0001	FEAT_CSV2_1p1 is implemented, but FEAT_CSV2_1p2 is not implemented.
0b0010	FEAT_CSV2_1p2 is implemented.

All other values are reserved.

FEAT_CSV2_1p1 implements the functionality identified by the value 0b0001.

FEAT_CSV2_1p2 implements the functionality identified by the value 0b0010.

In Armv8-R, the permitted values are 0b0000, 0b0001, and 0b0010.

The values 0b0001 and 0b0010 are permitted only when ID_AA64PFR0_EL1.CSV2 is 0b0001.

Access to this field is RO.

Bits [31:16]

Reserved, RES0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RAS_frac, bits [15:12]

RAS Extension fractional field.

The value of this field is an IMPLEMENTATION DEFINED choice of:

RAS_frac	Meaning
0b0000	If ID_AA64PFR0_EL1.RAS == 0b0001, support for the Reliability, Availability, and Serviceability Extension is implemented.
0b0001	If ID_AA64PFR0_EL1.RAS == 0b0001, as 0b0000 and adds support for: • Additional ERXMISC<m>_EL1 System registers. • Additional System registers ERXPFGCDN_EL1, ERXPFGCTL_EL1, and ERXPFGF_EL1, and the SCR_EL3.FIEN and HCR_EL2.FIEN trap controls, to support the optional RAS Common Fault Injection Model Extension. Error records accessed through System registers conform to RAS System Architecture v1.1, which includes simplifications to ERR<n>STATUS, and support for the optional RAS Timestamp and RAS Common Fault Injection Model Extensions.

All other values are reserved.

FEAT_RAS implements the functionality identified by the value 0b0000.

FEAT_RASv1p1 implements the functionality identified by the value 0b0001.

This field is valid only if ID_AA64PFR0_EL1.RAS == 0b0001.

Access to this field is RO.

Bits [11:8]

Reserved, RES0.

SSBS, bits [7:4]

Speculative Store Bypassing controls in AArch64 state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SSBS	Meaning
0b0000	AArch64 provides no mechanism to control the use of Speculative Store Bypassing.
0b0001	AArch64 provides the PSTATE.SSBS mechanism to mark regions that are Speculative Store Bypass Safe.
0b0010	As 0b0001, and adds the MSR and MRS instructions to directly read and write the PSTATE.SSBS field.

All other values are reserved.

FEAT_SSBS implements the functionality identified by the value 0b0001.

FEAT_SSBS2 implements the functionality identified by the value 0b0010.

Access to this field is RO.

Bits [3:0]

Reserved, RES0.

Accessing ID_AA64PFR1_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ID_AA64PFR1_EL1
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0100	0b001

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ID_AA64PFR1_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ID_AA64PFR1_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.53 ID_AA64PFR2_EL1, AArch64 Processor Feature Register 2

The ID_AA64PFR2_EL1 characteristics are:

Purpose

Provides additional information about implemented PE features in AArch64 state.

For general information about the interpretation of the ID registers, see 'Principles of the ID scheme for fields in ID registers'.

Configuration

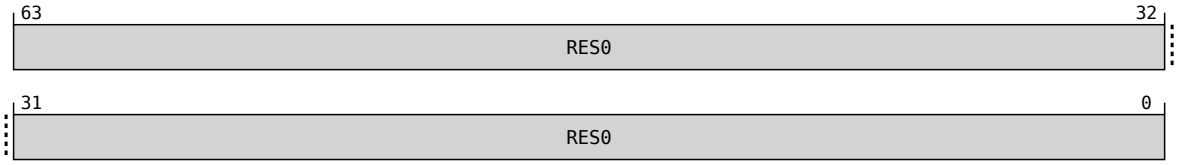
Note

Prior to the introduction of the features described by this register, this register was unnamed and reserved, RES0 from EL1 and EL2.

Attributes

ID_AA64PFR2_EL1 is a 64-bit register.

Field descriptions



Bits [63:0]

Reserved, RES0.

Accessing ID_AA64PFR2_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, ID_AA64PFR2_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0100	0b010

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if (!IsZero(ID_AA64PFR2_EL1) || boolean IMPLEMENTATION_DEFINED "ID_AA64PFR2_EL1 trapped by
        ↪HCR_EL2.TID3") && HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ID_AA64PFR2_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ID_AA64PFR2_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.54 ID_AFR0_EL1, AArch32 Auxiliary Feature Register 0

The ID_AFR0_EL1 characteristics are:

Purpose

Provides information about the IMPLEMENTATION DEFINED features of the PE in AArch32 state.

Must be interpreted with the Main ID Register, MIDR_EL1.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

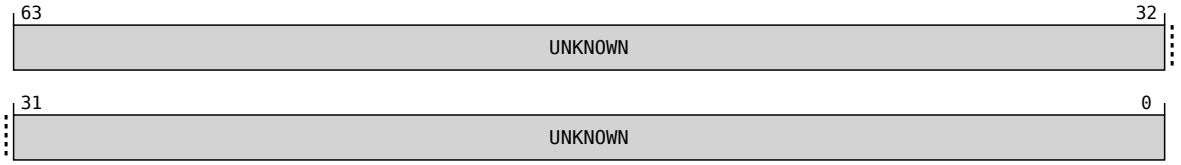
Configuration

There are no configuration notes.

Attributes

ID_AFR0_EL1 is a 64-bit register.

Field descriptions



Bits [63:0]

Reserved, UNKNOWN.

Accessing ID_AFR0_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ID_AFR0_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0001	0b011

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ID_AFR0_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ID_AFR0_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.55 ID_DFR0_EL1, AArch32 Debug Feature Register 0

The ID_DFR0_EL1 characteristics are:

Purpose

Provides top-level information about the debug system in AArch32 state.

Must be interpreted with the Main ID Register, MIDR_EL1.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

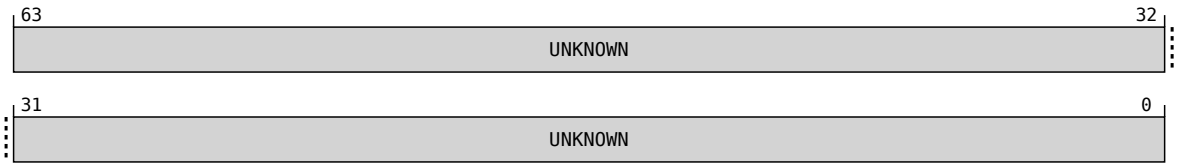
Configuration

There are no configuration notes.

Attributes

ID_DFR0_EL1 is a 64-bit register.

Field descriptions



Bits [63:0]

Reserved, UNKNOWN.

Accessing ID_DFR0_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, ID_DFR0_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0001	0b010

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ID_DFR0_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ID_DFR0_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.56 ID_DFR1_EL1, AArch32 Debug Feature Register 1

The ID_DFR1_EL1 characteristics are:

Purpose

Provides top-level information about the debug system in AArch32.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

Configuration

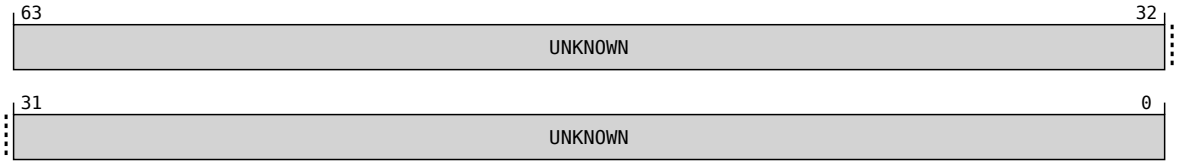
Note

Prior to the introduction of the features described by this register, this register was unnamed and reserved, RES0 from EL1 and EL2.

Attributes

ID_DFR1_EL1 is a 64-bit register.

Field descriptions



Bits [63:0]

Reserved, UNKNOWN.

Accessing ID_DFR1_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ID_DFR1_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0011	0b101

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if (!IsZero(ID_DFR1_EL1) || boolean IMPLEMENTATION_DEFINED "ID_DFR1_EL1 trapped by HCR_EL2.TID3") &&
        ↪HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ID_DFR1_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ID_DFR1_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.57 ID_ISAR0_EL1, AArch32 Instruction Set Attribute Register 0

The ID_ISAR0_EL1 characteristics are:

Purpose

Provides information about the instruction sets implemented by the PE in AArch32 state.

Must be interpreted with ID_ISAR1_EL1, ID_ISAR2_EL1, ID_ISAR3_EL1, ID_ISAR4_EL1, and ID_ISAR5_EL1.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

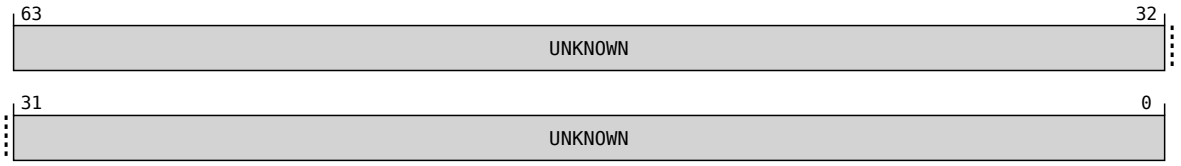
Configuration

There are no configuration notes.

Attributes

ID_ISAR0_EL1 is a 64-bit register.

Field descriptions



Bits [63:0]

Reserved, UNKNOWN.

Accessing ID_ISAR0_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, ID_ISAR0_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0010	0b000

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ID_ISAR0_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ID_ISAR0_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.58 ID_ISAR1_EL1, AArch32 Instruction Set Attribute Register 1

The ID_ISAR1_EL1 characteristics are:

Purpose

Provides information about the instruction sets implemented by the PE in AArch32 state.

Must be interpreted with ID_ISAR0_EL1, ID_ISAR2_EL1, ID_ISAR3_EL1, ID_ISAR4_EL1, and ID_ISAR5_EL1.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

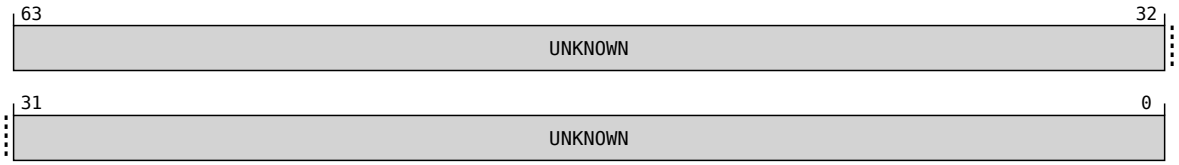
Configuration

There are no configuration notes.

Attributes

ID_ISAR1_EL1 is a 64-bit register.

Field descriptions



Bits [63:0]

Reserved, UNKNOWN.

Accessing ID_ISAR1_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, ID_ISAR1_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0010	0b001

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ID_ISAR1_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ID_ISAR1_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.59 ID_ISAR2_EL1, AArch32 Instruction Set Attribute Register 2

The ID_ISAR2_EL1 characteristics are:

Purpose

Provides information about the instruction sets implemented by the PE in AArch32 state.

Must be interpreted with ID_ISAR0_EL1, ID_ISAR1_EL1, ID_ISAR3_EL1, ID_ISAR4_EL1, and ID_ISAR5_EL1.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

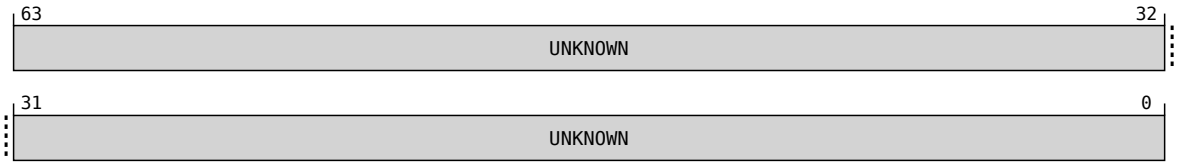
Configuration

There are no configuration notes.

Attributes

ID_ISAR2_EL1 is a 64-bit register.

Field descriptions



Bits [63:0]

Reserved, UNKNOWN.

Accessing ID_ISAR2_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, ID_ISAR2_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0010	0b010

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ID_ISAR2_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ID_ISAR2_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.60 ID_ISAR3_EL1, AArch32 Instruction Set Attribute Register 3

The ID_ISAR3_EL1 characteristics are:

Purpose

Provides information about the instruction sets implemented by the PE in AArch32 state.

Must be interpreted with ID_ISAR0_EL1, ID_ISAR1_EL1, ID_ISAR2_EL1, ID_ISAR4_EL1, and ID_ISAR5_EL1.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

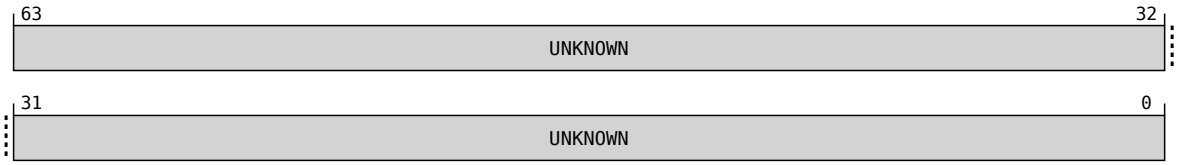
Configuration

There are no configuration notes.

Attributes

ID_ISAR3_EL1 is a 64-bit register.

Field descriptions



Bits [63:0]

Reserved, UNKNOWN.

Accessing ID_ISAR3_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, ID_ISAR3_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0010	0b011

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ID_ISAR3_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ID_ISAR3_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.61 ID_ISAR4_EL1, AArch32 Instruction Set Attribute Register 4

The ID_ISAR4_EL1 characteristics are:

Purpose

Provides information about the instruction sets implemented by the PE in AArch32 state.

Must be interpreted with ID_ISAR0_EL1, ID_ISAR1_EL1, ID_ISAR2_EL1, ID_ISAR3_EL1, and ID_ISAR5_EL1.

For general information about the interpretation of the ID registers, see 'Principles of the ID scheme for fields in ID registers'.

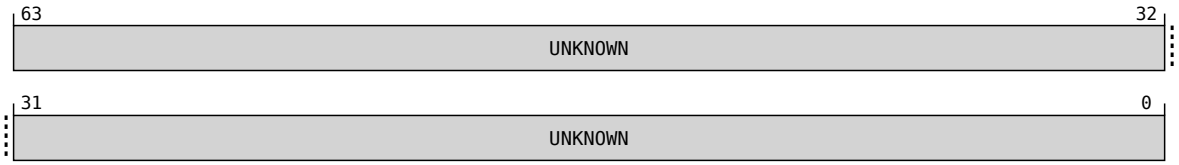
Configuration

There are no configuration notes.

Attributes

ID_ISAR4_EL1 is a 64-bit register.

Field descriptions



Bits [63:0]

Reserved, UNKNOWN.

Accessing ID_ISAR4_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, ID_ISAR4_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0010	0b100

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ID_ISAR4_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ID_ISAR4_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.62 ID_ISAR5_EL1, AArch32 Instruction Set Attribute Register 5

The ID_ISAR5_EL1 characteristics are:

Purpose

Provides information about the instruction sets implemented by the PE in AArch32 state.

Must be interpreted with ID_ISAR0_EL1, ID_ISAR1_EL1, ID_ISAR2_EL1, ID_ISAR3_EL1, and ID_ISAR4_EL1.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

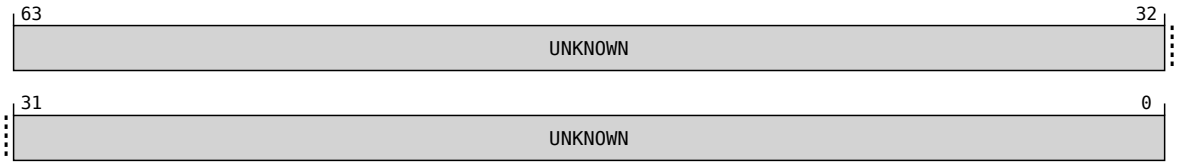
Configuration

There are no configuration notes.

Attributes

ID_ISAR5_EL1 is a 64-bit register.

Field descriptions



Bits [63:0]

Reserved, UNKNOWN.

Accessing ID_ISAR5_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, ID_ISAR5_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0010	0b101

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ID_ISAR5_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ID_ISAR5_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.63 ID_ISAR6_EL1, AArch32 Instruction Set Attribute Register 6

The ID_ISAR6_EL1 characteristics are:

Purpose

Provides information about the instruction sets implemented by the PE in AArch32 state.

Must be interpreted with ID_ISAR0_EL1, ID_ISAR1_EL1, ID_ISAR2_EL1, ID_ISAR3_EL1, ID_ISAR4_EL1 and ID_ISAR5_EL1.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

Configuration

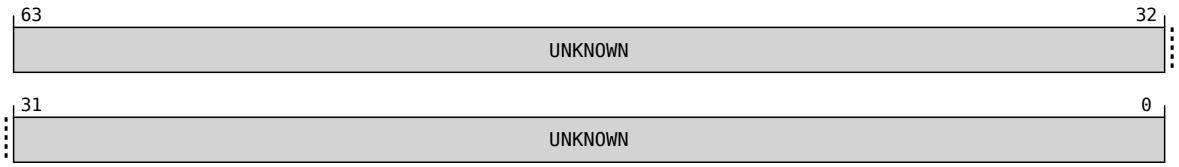
Note

Prior to the introduction of the features described by this register, this register was unnamed and reserved, RES0 from EL1 and EL2.

Attributes

ID_ISAR6_EL1 is a 64-bit register.

Field descriptions



Bits [63:0]

Reserved, UNKNOWN.

Accessing ID_ISAR6_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, ID_ISAR6_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0010	0b111

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if (!IsZero(ID_ISAR6_EL1) || boolean IMPLEMENTATION_DEFINED "ID_ISAR6_EL1 trapped by HCR_EL2.TID3")
        ↪&& HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ID_ISAR6_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ID_ISAR6_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.64 ID_MMFR0_EL1, AArch32 Memory Model Feature Register 0

The ID_MMFR0_EL1 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch32 state.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

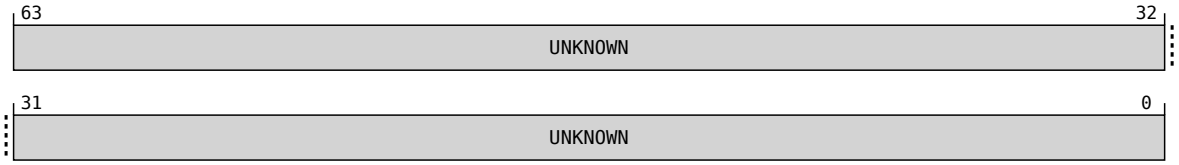
Configuration

There are no configuration notes.

Attributes

ID_MMFR0_EL1 is a 64-bit register.

Field descriptions



Bits [63:0]

Reserved, UNKNOWN.

Accessing ID_MMFR0_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, ID_MMFR0_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0001	0b100

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ID_MMFR0_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ID_MMFR0_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.65 ID_MMFR1_EL1, AArch32 Memory Model Feature Register 1

The ID_MMFR1_EL1 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch32 state.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

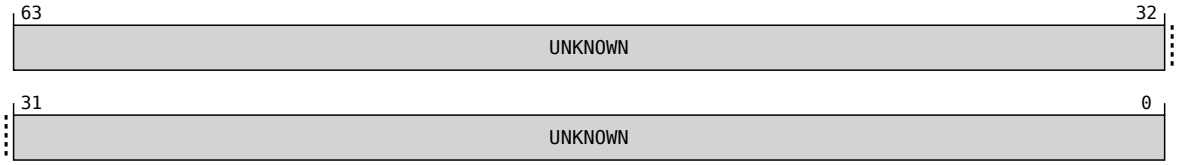
Configuration

There are no configuration notes.

Attributes

ID_MMFR1_EL1 is a 64-bit register.

Field descriptions



Bits [63:0]

Reserved, UNKNOWN.

Accessing ID_MMFR1_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ID_MMFR1_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0001	0b101

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ID_MMFR1_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ID_MMFR1_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.66 ID_MMFR2_EL1, AArch32 Memory Model Feature Register 2

The ID_MMFR2_EL1 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch32 state.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

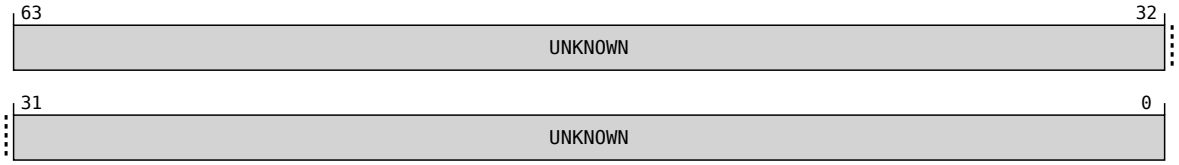
Configuration

There are no configuration notes.

Attributes

ID_MMFR2_EL1 is a 64-bit register.

Field descriptions



Bits [63:0]

Reserved, UNKNOWN.

Accessing ID_MMFR2_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ID_MMFR2_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0001	0b110

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ID_MMFR2_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ID_MMFR2_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.67 ID_MMFR3_EL1, AArch32 Memory Model Feature Register 3

The ID_MMFR3_EL1 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch32 state.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

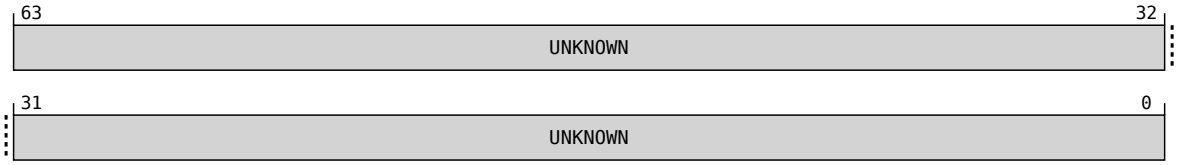
Configuration

There are no configuration notes.

Attributes

ID_MMFR3_EL1 is a 64-bit register.

Field descriptions



Bits [63:0]

Reserved, UNKNOWN.

Accessing ID_MMFR3_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, ID_MMFR3_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0001	0b111

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ID_MMFR3_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ID_MMFR3_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.68 ID_MMFR4_EL1, AArch32 Memory Model Feature Register 4

The ID_MMFR4_EL1 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch32 state.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

Configuration

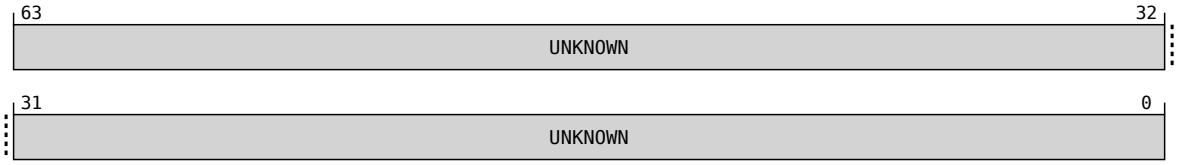
Note

Prior to the introduction of the features described by this register, this register was unnamed and reserved, RES0 from EL1 and EL2.

Attributes

ID_MMFR4_EL1 is a 64-bit register.

Field descriptions



Bits [63:0]

Reserved, UNKNOWN.

Accessing ID_MMFR4_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ID_MMFR4_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0010	0b110

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if (!IsZero(ID_MMFR4_EL1) || boolean IMPLEMENTATION_DEFINED "ID_MMFR4_EL1 trapped by HCR_EL2.TID3")
        ↪&& HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ID_MMFR4_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ID_MMFR4_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.69 ID_MMFR5_EL1, AArch32 Memory Model Feature Register 5

The ID_MMFR5_EL1 characteristics are:

Purpose

Provides information about the implemented memory model and memory management support in AArch32 state.

For general information about the interpretation of the ID registers, see 'Principles of the ID scheme for fields in ID registers'.

Configuration

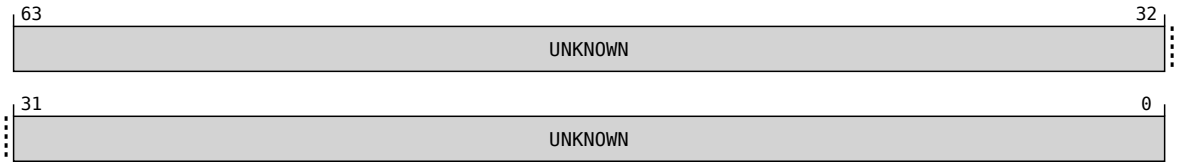
Note

Prior to the introduction of the features described by this register, this register was unnamed and reserved, RES0 from EL1, EL2, and EL3.

Attributes

ID_MMFR5_EL1 is a 64-bit register.

Field descriptions



Bits [63:0]

Reserved, UNKNOWN.

Accessing ID_MMFR5_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, ID_MMFR5_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0011	0b110

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if (!IsZero(ID_MMFR5_EL1) || boolean IMPLEMENTATION_DEFINED "ID_MMFR5_EL1 trapped by HCR_EL2.TID3")
        ↪&& HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ID_MMFR5_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ID_MMFR5_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.70 ID_PFR0_EL1, AArch32 Processor Feature Register 0

The ID_PFR0_EL1 characteristics are:

Purpose

Gives top-level information about the instruction sets supported by the PE in AArch32 state.

Must be interpreted with ID_PFR1_EL1.

For general information about the interpretation of the ID registers, see 'Principles of the ID scheme for fields in ID registers'.

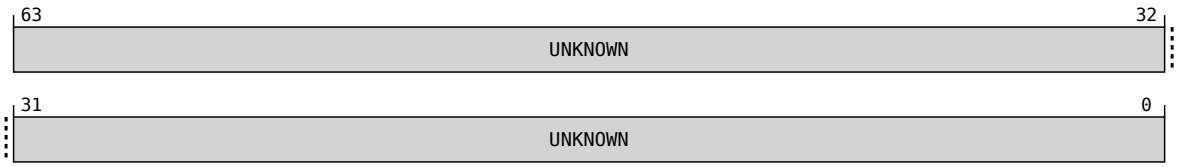
Configuration

There are no configuration notes.

Attributes

ID_PFR0_EL1 is a 64-bit register.

Field descriptions



Bits [63:0]

Reserved, UNKNOWN.

Accessing ID_PFR0_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, ID_PFR0_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0001	0b000

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ID_PFR0_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ID_PFR0_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.71 ID_PFR1_EL1, AArch32 Processor Feature Register 1

The ID_PFR1_EL1 characteristics are:

Purpose

Gives information about the AArch32 programmers' model.

Must be interpreted with ID_PFR0_EL1.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

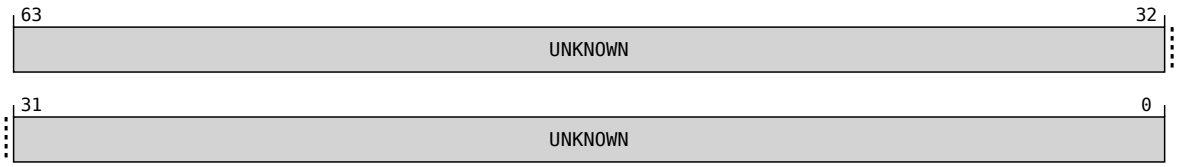
Configuration

There are no configuration notes.

Attributes

ID_PFR1_EL1 is a 64-bit register.

Field descriptions



Bits [63:0]

Reserved, UNKNOWN.

Accessing ID_PFR1_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, ID_PFR1_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0001	0b001

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ID_PFR1_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ID_PFR1_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.72 ID_PFR2_EL1, AArch32 Processor Feature Register 2

The ID_PFR2_EL1 characteristics are:

Purpose

Gives information about the AArch32 programmers' model.

Must be interpreted with ID_PFR0_EL1 and ID_PFR1_EL1.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

Configuration

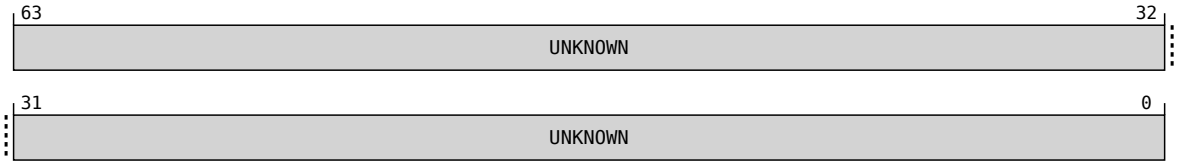
Note

Prior to the introduction of the features described by this register, this register was unnamed and reserved, RES0 from EL1 and EL2.

Attributes

ID_PFR2_EL1 is a 64-bit register.

Field descriptions



Bits [63:0]

Reserved, UNKNOWN.

Accessing ID_PFR2_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, ID_PFR2_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0011	0b100

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if (!IsZero(ID_PFR2_EL1) || boolean IMPLEMENTATION_DEFINED "ID_PFR2_EL1 trapped by HCR_EL2.TID3") &&
        ↪HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ID_PFR2_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ID_PFR2_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.73 ISR_EL1, Interrupt Status Register

The ISR_EL1 characteristics are:

Purpose

Shows the pending status of IRQ and FIQ interrupts and SError exceptions.

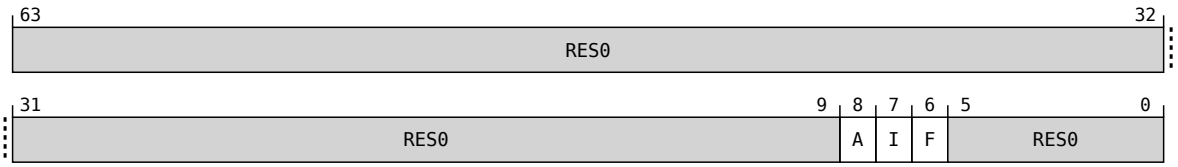
Configuration

There are no configuration notes.

Attributes

ISR_EL1 is a 64-bit register.

Field descriptions



Bits [63:9]

Reserved, RES0.

A, bit [8]

SError exception pending bit. Indicates whether an SError exception is pending.

A	Meaning
0b0	No pending SError.
0b1	An SError exception is pending.

If all of the following apply, then this field shows the pending status of virtual SError exceptions:

- HCR_EL2.AMO is 1.
- The PE is executing at EL1.

Otherwise, this field shows the pending status of physical SError exceptions.

If the physical SError exception is edge-triggered, this field is cleared to zero when the physical SError exception is taken.

I, bit [7]

IRQ pending bit. Indicates whether an IRQ interrupt is pending.

I	Meaning
0b0	No pending IRQ.
0b1	An IRQ interrupt is pending.

If all of the following apply, then this field shows the pending status of virtual IRQ interrupts:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- HCR_EL2.IMO is 1.
- The PE is executing at EL1.

Otherwise, this field shows the pending status of physical IRQ interrupts.

F, bit [6]

FIQ pending bit. Indicates whether an FIQ interrupt is pending.

F	Meaning
0b0	No pending FIQ.
0b1	An FIQ interrupt is pending.

If all of the following apply, then this field shows the pending status of virtual FIQ interrupts:

- HCR_EL2.FMO is 1.
- The PE is executing at EL1.

Otherwise, this field shows the pending status of physical FIQ interrupts.

Bits [5:0]

Reserved, RES0.

Accessing ISR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ISR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1100	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    X[t, 64] = ISR_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ISR_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.74 MAIR_EL1, Memory Attribute Indirection Register (EL1)

The MAIR_EL1 characteristics are:

Purpose

If VMSAv8-64 is enabled at stage 1 of EL1&0 translation regime, this register provides the memory attribute encodings corresponding to the possible AttrIndx values in a Long-descriptor format translation table entry for stage 1 translations at EL1.

If PMSAv8-64 is enabled at stage 1 of EL1&0 translation regime, this register provides the memory attribute encodings corresponding to the possible AttrIndx values in PRLAR_EL1 register for stage 1 translations.

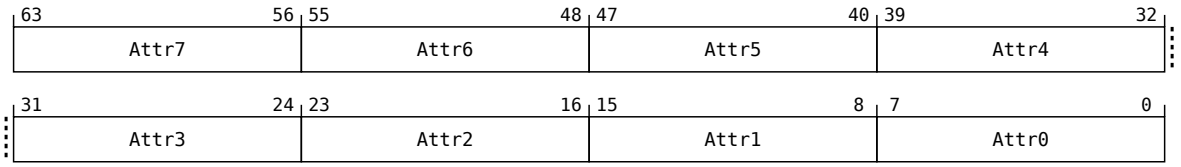
Configuration

There are no configuration notes.

Attributes

MAIR_EL1 is a 64-bit register.

Field descriptions



MAIR_EL1 is permitted to be cached in a TLB.

Attr<n> , bits [8n+7:8n], for n = 7 to 0

The memory attribute encoding for an AttrIndx[2:0] gives the value of <n> in Attr<n>.

Attr is encoded as follows:

Attr		Meaning
0b0000dd00		Device memory. See encoding of 'dd' for the type of Device memory.
0b0000ddxx	where xx != 00	UNPREDICTABLE.
0boooooiii	where oooo != 0000 and iiii != 0000	Normal memory. See encoding of 'oooo' and 'iiii' for the type of Normal memory.
0bxxxx0000	where xxxx != 0000	UNPREDICTABLE.

dd is encoded as follows:

'dd'	Meaning
0b00	Device-nGnRnE memory.
0b01	Device-nGnRE memory.
0b10	Device-nGRE memory.
0b11	Device-GRE memory.

oooo is encoded as follows:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

'oooo'		Meaning
0b0000		See encoding of Attr.
0b00RW	where RW != 00	Normal memory, Outer Write-Through Transient.
0b0100		Normal memory, Outer Non-cacheable.
0b01RW	where RW != 00	Normal memory, Outer Write-Back Transient.
0b10RW		Normal memory, Outer Write-Through Non-transient.
0b11RW		Normal memory, Outer Write-Back Non-transient.

R encodes the Outer Read-Allocate policy and W encodes the Outer Write-Allocate policy.

iiii is encoded as follows:

'iiii'		Meaning
0b0000		See encoding of Attr.
0b00RW	where RW != 00	Normal memory, Inner Write-Through Transient.
0b0100		Normal memory, Inner Non-cacheable.
0b01RW	where RW != 00	Normal memory, Inner Write-Back Transient.
0b10RW		Normal memory, Inner Write-Through Non-transient.
0b11RW		Normal memory, Inner Write-Back Non-transient.

R encodes the Inner Read-Allocate policy and W encodes the Inner Write-Allocate policy.

In oooo and iiii, R and W are encoded as follows:

'R' or 'W'	Meaning
0b0	No Allocate.
0b1	Allocate.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing MAIR_EL1

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, MAIR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = MAIR_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = MAIR_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR MAIR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        MAIR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    MAIR_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.75 MAIR_EL2, Memory Attribute Indirection Register (EL2)

The MAIR_EL2 characteristics are:

Purpose

Provides the memory attribute encodings corresponding to the possible AttrIndx values in PRLAR_EL2 for stage 1 EL2 translation regime and for stage 2 EL1&0 translation regime.

For stage 2 EL1&0 translations, the memory attributes are derived from MAIR_EL2 register as described in the Armv8-R AArch64 architecture.

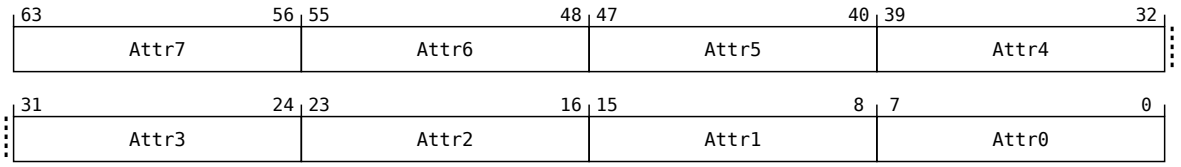
Configuration

There are no configuration notes.

Attributes

MAIR_EL2 is a 64-bit register.

Field descriptions



MAIR_EL2 is permitted to be cached in a TLB.

Attr<n> , bits [8n+7:8n], for n = 7 to 0

The memory attribute encoding for an AttrIndx[2:0] gives the value of <n> in Attr<n>.

Attr is encoded as follows:

Attr		Meaning
0b0000dd00		Device memory. See encoding of 'dd' for the type of Device memory.
0b0000ddxx	where xx != 00	UNPREDICTABLE.
0boooiiii	where oooo != 0000 and iiii != 0000	Normal memory. See encoding of 'oooo' and 'iiii' for the type of Normal memory.
0bxxxx0000	where xxxx != 0000	UNPREDICTABLE.

dd is encoded as follows:

'dd'	Meaning
0b00	Device-nGnRnE memory.
0b01	Device-nGnRE memory.
0b10	Device-nGRE memory.
0b11	Device-GRE memory.

oooo is encoded as follows:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

'oooo'		Meaning
0b0000		See encoding of Attr.
0b00RW	where RW != 00	Normal memory, Outer Write-Through Transient.
0b0100		Normal memory, Outer Non-cacheable.
0b01RW	where RW != 00	Normal memory, Outer Write-Back Transient.
0b10RW		Normal memory, Outer Write-Through Non-transient.
0b11RW		Normal memory, Outer Write-Back Non-transient.

R encodes the Outer Read-Allocate policy and W encodes the Outer Write-Allocate policy.

iiii is encoded as follows:

'iiii'		Meaning
0b0000		See encoding of Attr.
0b00RW	where RW != 00	Normal memory, Inner Write-Through Transient.
0b0100		Normal memory, Inner Non-cacheable.
0b01RW	where RW != 00	Normal memory, Inner Write-Back Transient.
0b10RW		Normal memory, Inner Write-Through Non-transient.
0b11RW		Normal memory, Inner Write-Back Non-transient.

R encodes the Inner Read-Allocate policy and W encodes the Inner Write-Allocate policy.

In oooo and iiii, R and W are encoded as follows:

'R' or 'W'	Meaning
0b0	No Allocate.
0b1	Allocate.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing MAIR_EL2

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, MAIR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = MAIR_EL2;
```

MSR MAIR_EL2, <Xt>

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b100	0b1010	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    MAIR_EL2 = X[t, 64];
```

MRS <Xt>, MAIR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = MAIR_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = MAIR_EL1;
```

MSR MAIR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1010	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        MAIR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    MAIR_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.76 MIDR_EL1, Main ID Register

The MIDR_EL1 characteristics are:

Purpose

Provides identification information for the PE, including an implementer code for the device and a device ID number.

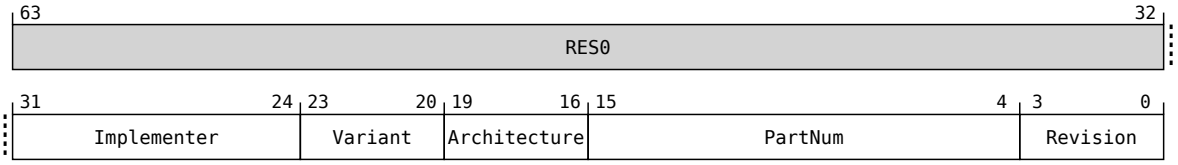
Configuration

AArch64 System register MIDR_EL1 bits [31:0] are architecturally mapped to External register MIDR_EL1[31:0].

Attributes

MIDR_EL1 is a 64-bit register.

Field descriptions



Bits [63:32]

Reserved, RES0.

Implementer, bits [31:24]

The Implementer code. This field must hold an implementer code that has been assigned by Arm.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Implementer	Meaning
0x00	Reserved for software use.
0x41	Arm Limited.
0x42	Broadcom Corporation.
0x43	Cavium Inc.
0x44	Digital Equipment Corporation.
0x46	Fujitsu Ltd.
0x49	Infineon Technologies AG.
0x4D	Motorola or Freescale Semiconductor Inc.
0x4E	NVIDIA Corporation.
0x50	Applied Micro Circuits Corporation.
0x51	Qualcomm Inc.
0x56	Marvell International Ltd.
0x69	Intel Corporation.
0xC0	Ampere Computing.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Arm can assign codes that are not published in this manual. All values not assigned by Arm are reserved and must not be used.

Access to this field is RO.

Variant, bits [23:20]

Variant number. Typically, this field is used to distinguish between different product variants, or major revisions of a product.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Architecture, bits [19:16]

Architecture version.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Architecture	Meaning
0b0001	Armv4.
0b0010	Armv4T.
0b0011	Armv5 (obsolete).
0b0100	Armv5T.
0b0101	Armv5TE.
0b0110	Armv5TEJ.
0b0111	Armv6.
0b1111	Architectural features are individually identified in the ID_* registers.

All other values are reserved.

Access to this field is RO.

PartNum, bits [15:4]

Primary Part Number for the device.

On processors implemented by Arm, if the top four bits of the primary part number are 0x0 or 0x7, the variant and architecture are encoded differently.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Revision, bits [3:0]

Revision number for the device.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing MIDR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, MIDR_EL1
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0000	0b000

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    X[t, 64] = VPIDR_EL2;
elsif PSTATE.EL == EL2 then
    X[t, 64] = MIDR_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.77 MPIDR_EL1, Multiprocessor Affinity Register

The MPIDR_EL1 characteristics are:

Purpose

In a multiprocessor system, provides an additional PE identification mechanism.

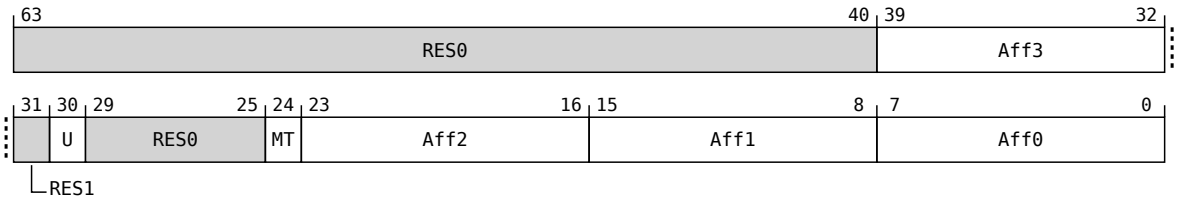
Configuration

In a uniprocessor system, Arm recommends that each Aff<n> field of this register returns a value of 0.

Attributes

MPIDR_EL1 is a 64-bit register.

Field descriptions



Bits [63:40]

Reserved, RES0.

Aff3, bits [39:32]

Affinity level 3. See the description of Aff0 for more information.

Aff3 is not supported in AArch32 state.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Bit [31]

Reserved, RES1.

U, bit [30]

Indicates a Uniprocessor system, as distinct from PE 0 in a multiprocessor system.

The value of this field is an IMPLEMENTATION DEFINED choice of:

U	Meaning
0b0	Processor is part of a multiprocessor system.
0b1	Processor is part of a uniprocessor system.

Access to this field is RO.

Bits [29:25]

Reserved, RES0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MT, bit [24]

Indicates whether the lowest level of affinity consists of logical PEs that are implemented using an interdependent approach, such as multithreading. See the description of Aff0 for more information about affinity levels.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MT	Meaning
0b0	Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is largely independent.
0b1	Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is very interdependent.

Note

This field does not indicate that multithreading is implemented and does not indicate that PEs with different affinity level 0 values, and the same values for affinity level 1 and higher are implemented.

Access to this field is RO.

Aff2, bits [23:16]

Affinity level 2. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Aff1, bits [15:8]

Affinity level 1. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Aff0, bits [7:0]

Affinity level 0. The value of the MPIDR.{Aff2, Aff1, Aff0} or MPIDR_EL1.{Aff3, Aff2, Aff1, Aff0} set of fields of each PE must be unique within the system as a whole.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing MPIDR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, MPIDR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0000	0b101

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    X[t, 64] = VMPIDR_EL2;
elsif PSTATE.EL == EL2 then
    X[t, 64] = MPIDR_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.78 MVFR0_EL1, AArch32 Media and VFP Feature Register 0

The MVFR0_EL1 characteristics are:

Purpose

Describes the features provided by the AArch32 Advanced SIMD and floating-point implementation.

Must be interpreted with MVFR1_EL1 and MVFR2_EL1.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

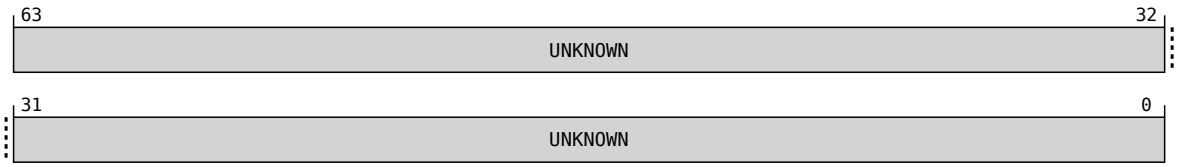
Configuration

In an implementation where at least one Exception level supports execution in AArch32 state, but there is no support for Advanced SIMD and floating-point operation, this register is RAZ.

Attributes

MVFR0_EL1 is a 64-bit register.

Field descriptions



Bits [63:0]

Reserved, UNKNOWN.

Accessing MVFR0_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, MVFR0_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0011	0b000

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = MVFR0_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = MVFR0_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.79 MVFR1_EL1, AArch32 Media and VFP Feature Register 1

The MVFR1_EL1 characteristics are:

Purpose

Describes the features provided by the AArch32 Advanced SIMD and floating-point implementation.

Must be interpreted with MVFR0_EL1 and MVFR2_EL1.

For general information about the interpretation of the ID registers see 'Principles of the ID scheme for fields in ID registers'.

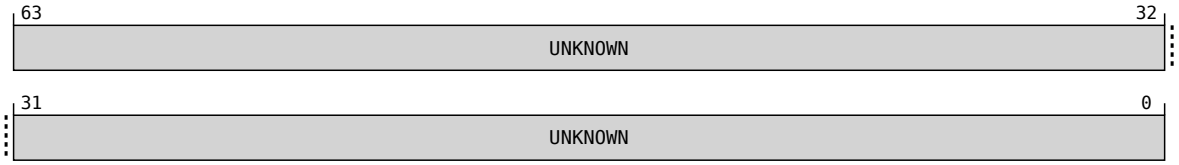
Configuration

In an implementation where at least one Exception level supports execution in AArch32 state, but there is no support for Advanced SIMD and floating-point operation, this register is RAZ.

Attributes

MVFR1_EL1 is a 64-bit register.

Field descriptions



Bits [63:0]

Reserved, UNKNOWN.

Accessing MVFR1_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, MVFR1_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0011	0b001

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = MVFR1_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = MVFR1_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.80 MVFR2_EL1, AArch32 Media and VFP Feature Register 2

The MVFR2_EL1 characteristics are:

Purpose

Describes the features provided by the AArch32 Advanced SIMD and floating-point implementation.

Must be interpreted with MVFR0_EL1 and MVFR1_EL1.

For general information about the interpretation of the ID registers, see 'Principles of the ID scheme for fields in ID registers'.

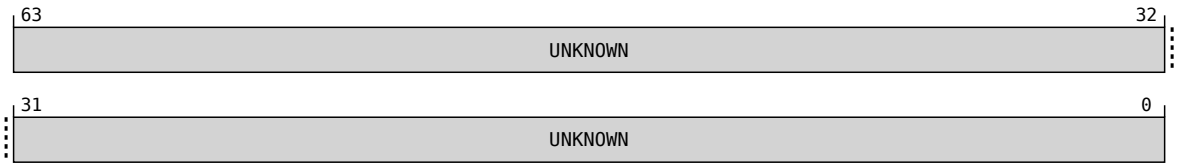
Configuration

In an implementation where at least one Exception level supports execution in AArch32 state, but there is no support for Advanced SIMD and floating-point operation, this register is RAZ.

Attributes

MVFR2_EL1 is a 64-bit register.

Field descriptions



Bits [63:0]

Reserved, UNKNOWN.

Accessing MVFR2_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, MVFR2_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0011	0b010

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = MVFR2_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = MVFR2_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.81 PAR_EL1, Physical Address Register

The PAR_EL1 characteristics are:

Purpose

Returns the output address (OA) from an Address translation instruction that executed successfully, or fault information if the instruction did not execute successfully.

Configuration

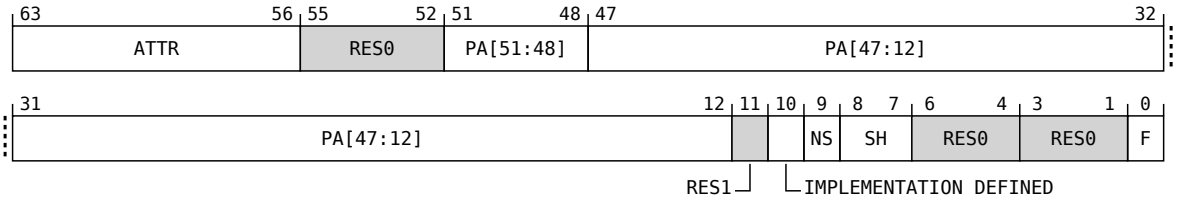
There are no configuration notes.

Attributes

PAR_EL1 is a 64-bit register.

Field descriptions

When GetPAR_EL1_F() == '0':



This section describes the register value returned by the successful execution of an Address translation instruction. Software might subsequently write a different value to the register, and that write does not affect the operation of the PE.

On a successful conversion, the PAR_EL1 can return a value that indicates the resulting attributes, rather than the values that appear in the Translation table descriptors. More precisely:

- The PAR_EL1.{ATTR, SH} fields are permitted to report the resulting attributes, as determined by any permitted implementation choices and any applicable configuration bits, instead of reporting the values that appear in the Translation table descriptors.
- See the PAR_EL1.NS bit description for constraints on the value it returns.

ATTR, bits [63:56]

Memory attributes for the returned output address. This field uses the same encoding as the Attr<n> fields in MAIR_EL1, and MAIR_EL2.

The value returned in this field can be the resulting attribute that is actually implemented by the implementation, as determined by any permitted implementation choices and any applicable configuration bits, instead of the value that appears in the Translation table descriptor.

Note

The attributes presented are consistent with the stages of translation applied in the address translation instruction. If the instruction performed a stage 1 translation only, the attributes are from the stage 1 translation. If the instruction performed a stage 1 and stage 2 translation, the attributes are from the combined stage 1 and stage 2 translation.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [55:52, 6:4]

Reserved, RES0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PA[51:48], bits [51:48]

When FEAT_LPA is implemented:

Extension to PA[47:12]. For more information, see PA[47:12].

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PA[47:12], bits [47:12]

Output address. The output address (OA) corresponding to the supplied input address. This field returns address bits[47:12].

When FEAT_LPA is implemented and 52-bit addresses are in use, PA[51:48] forms the upper part of the address value. Otherwise, when 52-bit addresses are not in use, PA[51:48] is RES0.

For implementations with fewer than 48 physical address bits, the corresponding upper bits in this field are RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [11]

Reserved, RES1.

IMPLEMENTATION DEFINED, bit [10]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

NS, bit [9]

Non-secure. The NS attribute for a translation table entry from a Secure translation regime.

For a result from a Secure translation regime, when SCR_EL3.EEL2 is 1, this bit distinguishes between the Secure and Non-secure intermediate physical address space of the translation for the instructions:

- In AArch64 state: AT S1E1R, AT S1E1W, AT S1E1RP, AT S1E1WP, AT S1E0R, and AT S1E0W.

Otherwise, this bit reflects the Security state of the physical address space of the translation. This means it reflects the effect of the NSTable bits of earlier levels of the translation table walk if those NSTable bits have an effect on the translation.

For a result from a Non-secure translation regime, this bit is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SH, bits [8:7]

Shareability attribute, for the returned output address.

SH	Meaning
0b00	Non-shareable.
0b10	Outer Shareable.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SH	Meaning
0b11	Inner Shareable.

The value 0b01 is reserved.

Note

This field returns the value 0b10 for:

- Any type of Device memory.
- Normal memory with both Inner Non-cacheable and Outer Non-cacheable attributes.

The value returned in this field can be the resulting attribute, as determined by any permitted implementation choices and any applicable configuration bits, instead of the value that appears in the Translation table descriptor.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [3:1]

Reserved, RES0.

F, bit [0]

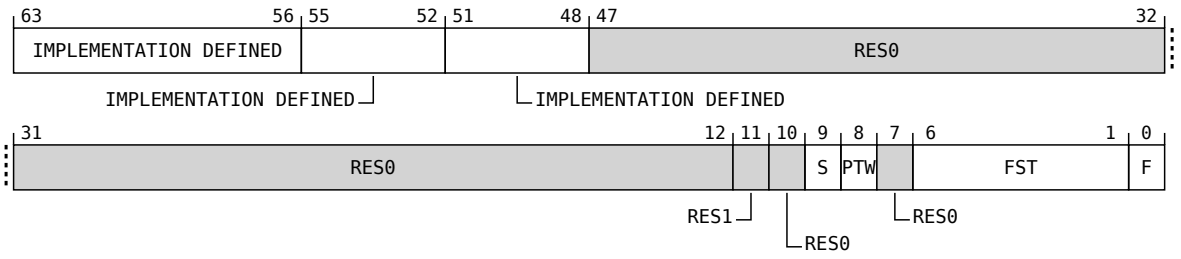
Indicates whether the instruction performed a successful address translation.

F	Meaning
0b0	Address translation completed successfully.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When GetPAR_EL1_F() == '1':



This section describes the register value returned by a fault on the execution of an Address translation instruction. Software might subsequently write a different value to the register, and that write does not affect the operation of the PE.

IMPLEMENTATION DEFINED, bits [63:56]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IMPLEMENTATION DEFINED, bits [55:52]

IMPLEMENTATION DEFINED.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IMPLEMENTATION DEFINED, bits [51:48]

IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [47:12]

Reserved, RES0.

Bit [11]

Reserved, RES1.

Bit [10]

Reserved, RES0.

S, bit [9]

Indicates the translation stage at which the translation aborted:

S	Meaning
0b0	Translation aborted because of a fault in the stage 1 translation.
0b1	Translation aborted because of a fault in the stage 2 translation.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PTW, bit [8]

If this bit is set to 1, it indicates the translation aborted because of a stage 2 fault during a stage 1 translation table walk.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [7]

Reserved, RES0.

FST, bits [6:1]

Fault status code, as shown in the Data Abort exception ESR encoding.

FST	Meaning	Applies when
0b000000	Address size fault, level 0 of translation or translation table base register.	
0b000001	Address size fault, level 1.	
0b000010	Address size fault, level 2.	
0b000011	Address size fault, level 3.	

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FST	Meaning	Applies when
0b000100	Translation fault, level 0.	
0b000101	Translation fault, level 1.	
0b000110	Translation fault, level 2.	
0b000111	Translation fault, level 3.	
0b001001	Access flag fault, level 1.	
0b001010	Access flag fault, level 2.	
0b001011	Access flag fault, level 3.	
0b001100	Permission fault, level 0.	
0b001101	Permission fault, level 1.	
0b001110	Permission fault, level 2.	
0b001111	Permission fault, level 3.	
0b011100	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 0.	FEAT_RAS is not implemented
0b011101	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 1.	FEAT_RAS is not implemented
0b011110	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 2.	FEAT_RAS is not implemented
0b011111	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level 3.	FEAT_RAS is not implemented
0b110000	TLB conflict abort.	
0b110001	Unsupported atomic hardware update fault.	FEAT_HAFDBS is implemented

Note

The encodings for FST do not include Synchronous External abort on translation table walk or hardware update of translation table, because these MMU faults are reported as a Data Abort exception instead of being recorded in PAR_EL1. See Exceptions to reporting the fault in PAR_EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [0]

Indicates whether the instruction performed a successful address translation.

F	Meaning
0b1	Address translation aborted.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PAR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PAR_EL1
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b000	0b0111	0b0100	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    X[t, 64] = PAR_EL1<63:0>;
elsif PSTATE.EL == EL2 then
    X[t, 64] = PAR_EL1<63:0>;
```

MSR PAR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0111	0b0100	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    PAR_EL1<63:0> = X[t, 64];
elsif PSTATE.EL == EL2 then
    PAR_EL1<63:0> = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.82 REVIDR_EL1, Revision ID Register

The REVIDR_EL1 characteristics are:

Purpose

Provides implementation-specific minor revision information.

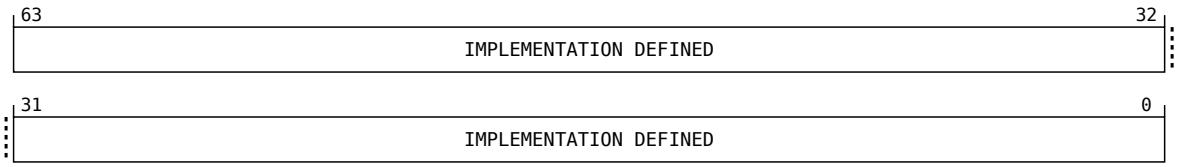
Configuration

If REVIDR_EL1 has the same value as MIDR_EL1, then its contents have no significance.

Attributes

REVIDR_EL1 is a 64-bit register.

Field descriptions



IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

Accessing REVIDR_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, REVIDR_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0000	0b110

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TID1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = REVIDR_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = REVIDR_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.83 RMR_EL2, Reset Management Register (EL2)

The RMR_EL2 characteristics are:

Purpose

When this register is implemented:

- A write to the register at EL2 can request a Warm reset.
- If EL2 can use all Execution states, this register specifies the Execution state that the PE boots into on a Warm reset.

Configuration

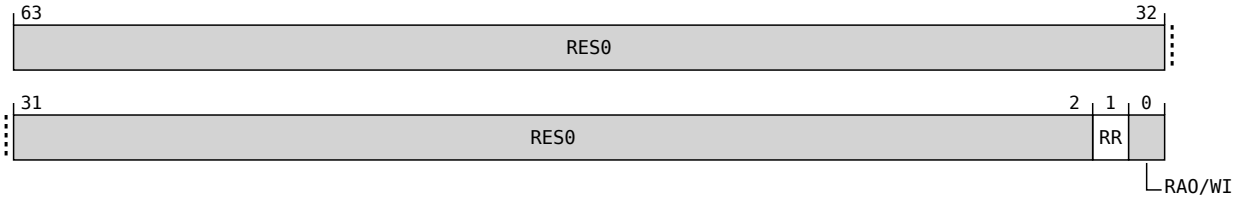
When EL2 is the highest implemented Exception level:

- If EL2 can use all Execution states then this register must be implemented.
- In an AArch64 only implementation it is IMPLEMENTATION DEFINED whether the register is implemented.

Attributes

RMR_EL2 is a 64-bit register.

Field descriptions



Bits [63:2]

Reserved, RES0.

RR, bit [1]

Reset Request. Setting this bit to 1 requests a Warm reset.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Bit [0]

Reserved, RAO/WI.

Accessing RMR_EL2

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, RMR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b1100	0b0000	0b010

```
if PSTATE.EL == EL2 then
    X[t, 64] = RMR_EL2;
else
    UNDEFINED;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR RMR_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1100	0b0000	0b010

```
if PSTATE.EL == EL2 then
    RMR_EL2 = X[t, 64];
else
    UNDEFINED;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.84 RVBAR_EL2, Reset Vector Base Address Register (if EL3 not implemented)

The RVBAR_EL2 characteristics are:

Purpose

If EL2 is the highest Exception level implemented, contains the IMPLEMENTATION DEFINED address that execution starts from after reset when executing in AArch64 state.

Configuration

There are no configuration notes.

Attributes

RVBAR_EL2 is a 64-bit register.

Field descriptions

ResetAddress, bits [63:0]

The IMPLEMENTATION DEFINED address that execution starts from after reset when executing in 64-bit state. Bits[1:0] of this register are 00, as this address must be aligned, and the address must be within the physical address size supported by the PE.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing RVBAR_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, RVBAR_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1100	0b0000	0b001

```
if PSTATE.EL == EL2 then
    X[t, 64] = RVBAR_EL2;
else
    UNDEFINED;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.85 S3_<op1>_<Cn>_<Cm>_<op2>, IMPLEMENTATION DEFINED Registers

The S3_<op1><*Cn*><Cm>_<op2> characteristics are:

Purpose

This area of the instruction set space is reserved for IMPLEMENTATION DEFINED registers.

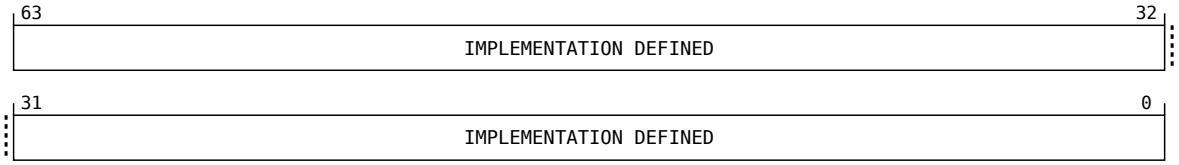
Configuration

There are no configuration notes.

Attributes

S3_<op1><*Cn*><Cm>_<op2> is a 64-bit register.

Field descriptions



IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

Accessing S3_<op1>_<Cn>_<Cm>_<op2>

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, S3_<op1>_C<Cn>_C<Cm>_<op2>`

op0	op1	CRn	CRm	op2
0b11	op1[2:0]	0b1x11	Cm[3:0]	op2[2:0]

```
if PSTATE.EL == EL0 then
    AArch64.ImpDefSysRegRead(op0, op1, CRn, CRm, op2, t);
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TIDCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.ImpDefSysRegRead(op0, op1, CRn, CRm, op2, t);
elsif PSTATE.EL == EL2 then
    AArch64.ImpDefSysRegRead(op0, op1, CRn, CRm, op2, t);
```

`MSR S3_<op1>_C<Cn>_C<Cm>_<op2>, <Xt>`

op0	op1	CRn	CRm	op2
0b11	op1[2:0]	0b1x11	Cm[3:0]	op2[2:0]

```
if PSTATE.EL == EL0 then
    AArch64.ImpDefSysRegWrite(op0, op1, CRn, CRm, op2, t);
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TIDCP == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        AArch64.ImpDefSysRegWrite(op0, op1, CRn, CRm, op2, t);
elsif PSTATE.EL == EL2 then
    AArch64.ImpDefSysRegWrite(op0, op1, CRn, CRm, op2, t);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.86 SCTLR_EL1, System Control Register (EL1)

The SCTLR_EL1 characteristics are:

Purpose

Provides top-level control of the system, including its memory system, at EL1 and EL0.

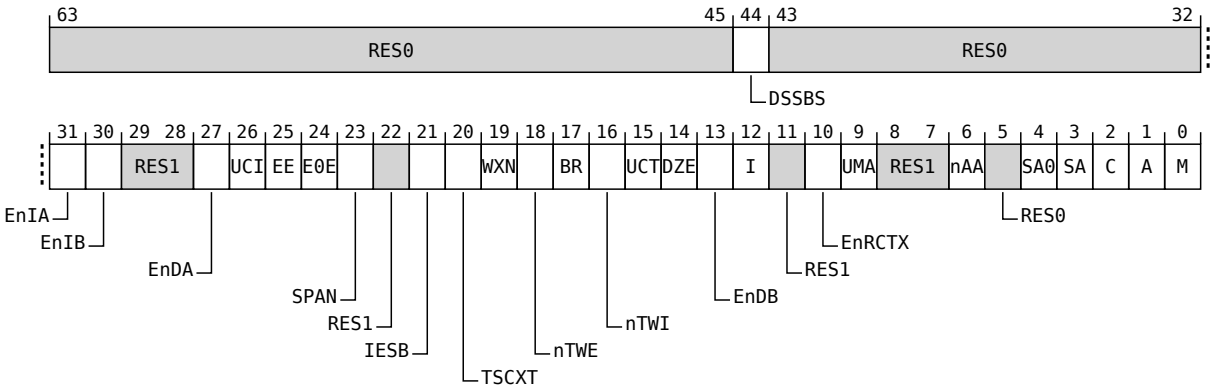
Configuration

There are no configuration notes.

Attributes

SCTLR_EL1 is a 64-bit register.

Field descriptions



Bits [63:45]

Reserved, RES0.

DSSBS, bit [44]

When FEAT_SSBS is implemented:

Default PSTATE.SSBS value on Exception Entry.

DSSBS	Meaning
0b0	PSTATE.SSBS is set to 0 on an exception to EL1.
0b1	PSTATE.SSBS is set to 1 on an exception to EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

Otherwise:

Reserved, RES0.

Bits [43:32]

Reserved, RES0.

EnIA, bit [31]

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_PAuth is implemented:

Controls enabling of pointer authentication of instruction addresses, using the APIAKey_EL1 key, at EL1 and at EL0 in the EL1&0 translation regime.

EnIA	Meaning
0b0	Pointer authentication of instruction addresses at EL1 and at EL0 in the EL1&0 translation regime, using the APIAKey_EL1 key, is not enabled.
0b1	Pointer authentication of instruction addresses at EL1 and at EL0 in the EL1&0 translation regime, using the APIAKey_EL1 key, is enabled.

Note

This field controls the behavior of the AddPACIA and AuthIA pseudocode functions. When pointer authentication is enabled, AddPACIA returns a copy of a pointer to which a pointer authentication code has been added, and AuthIA returns an authenticated copy of a pointer. When pointer authentication is disabled, both of these functions are NOP.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EnIB, bit [30]

When FEAT_PAuth is implemented:

Controls enabling of pointer authentication of instruction addresses, using the APIBKey_EL1 key, at EL1 and at EL0 in the EL1&0 translation regime.

EnIB	Meaning
0b0	Pointer authentication of instruction addresses at EL1 and at EL0 in the EL1&0 translation regime, using the APIBKey_EL1 key, is not enabled.
0b1	Pointer authentication of instruction addresses at EL1 and at EL0 in the EL1&0 translation regime, using the APIBKey_EL1 key, is enabled.

Note

This field controls the behavior of the AddPACIB and AuthIB pseudocode functions. When pointer authentication is enabled, AddPACIB returns a copy of a pointer to which a pointer authentication code has been added, and AuthIB returns an authenticated copy of a pointer. When pointer authentication is disabled, both of these functions are NOP.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [29:28]

Reserved, RES1.

EnDA, bit [27]

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_PAuth is implemented:

Controls enabling of pointer authentication of data addresses, using the APDAKey_EL1 key, at EL1 and at EL0 in the EL1&0 translation regime.

EnDA	Meaning
0b0	Pointer authentication of data addresses at EL1 and at EL0 in the EL1&0 translation regime, using the APDAKey_EL1 key, is not enabled.
0b1	Pointer authentication of data addresses at EL1 and at EL0 in the EL1&0 translation regime, using the APDAKey_EL1 key, is enabled.

Note

This field controls the behavior of the AddPACDA and AuthDA pseudocode functions. When pointer authentication is enabled, AddPACDA returns a copy of a pointer to which a pointer authentication code has been added, and AuthDA returns an authenticated copy of a pointer. When pointer authentication is disabled, both of these functions are NOP.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

UCI, bit [26]

Traps EL0 execution of cache maintenance instructions, to EL1, or to EL2 when HCR_EL2.TGE is 1, from AArch64 state only, reported using EC syndrome value 0x18, as follows:

- DC CVAU, DC CIVAC, DC CVAC, and IC IVAU.
- If FEAT_DPB is implemented, DC CVAP.
- If FEAT_DPB2 is implemented, DC CVADP.

UCI	Meaning
0b0	For each of the specified instructions, if the execution of the instruction can be trapped, access at EL0 using AArch64 is trapped.
0b1	This control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EE, bit [25]

When FEAT_MixedEnd is implemented:

Endianness of data accesses at EL1, and stage 1 translation table walks in the EL1&0 translation regime.

EE	Meaning
0b0	Explicit data accesses at EL1, and stage 1 translation table walks in the EL1&0 translation regime are little-endian.
0b1	Explicit data accesses at EL1, and stage 1 translation table walks in the EL1&0 translation regime are big-endian.

The EE bit is permitted to be cached in a TLB.

The reset behavior of this field is:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

When FEAT_BigEnd is implemented:

Explicit data accesses at EL1, and stage 1 translation table walks in the EL1&0 translation regime are big-endian.

Reserved, RES1.

Otherwise:

Explicit data accesses at EL1, and stage 1 translation table walks in the EL1&0 translation regime are little-endian.

Reserved, RES0.

E0E, bit [24]

When FEAT_MixedEndEL0 is implemented:

Endianness of data accesses at EL0.

E0E	Meaning
0b0	Explicit data accesses at EL0 are little-endian.
0b1	Explicit data accesses at EL0 are big-endian.

This bit has no effect on the endianness of Explicit Memory Effects generated by unprivileged memory access instructions executed at EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_BigEndEL0 is implemented:

Explicit data accesses at EL0 are big-endian.

Reserved, RES1.

Otherwise:

Explicit data accesses at EL0 are little-endian.

Reserved, RES0.

SPAN, bit [23]

When FEAT_PAN is implemented:

Set Privileged Access Never, on taking an exception to EL1.

SPAN	Meaning
0b0	PSTATE.PAN is set to 1 on taking an exception to EL1.
0b1	The value of PSTATE.PAN is left unchanged on taking an exception to EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES1.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bit [22]

Reserved, RES1.

IESB, bit [21]

When FEAT_IESB is implemented:

Implicit Error Synchronization event enable. Possible values are:

IESB	Meaning
0b0	Disabled.
0b1	An implicit error synchronization event is added: • At each exception taken to EL1. • Before the operational pseudocode of each Exception Return instruction executed at EL1.

When the PE is in Debug state, the effect of this field is CONSTRAINED UNPREDICTABLE, and its Effective value might be 0 or 1 regardless of the value of the field. If the Effective value of the field is 1, then an implicit error synchronization event is added after each DCPSx instruction taken to EL1 and before each DRPS instruction executed at EL1, in addition to the other cases where it is added.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TSCXT, bit [20]

When FEAT_CSV2_2 is implemented or FEAT_CSV2_1p2 is implemented:

Trap EL0 Access to the SCXTNUM_EL0 register, when EL0 is using AArch64.

TSCXT	Meaning
0b0	EL0 access to SCXTNUM_EL0 is not disabled by this mechanism.
0b1	EL0 access to SCXTNUM_EL0 is disabled, causing an exception to EL1, or to EL2 whenHCR_EL2.TGE is 1. The value of SCXTNUM_EL0 is treated as 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES1.

WXN, bit [19]

Write permission implies XN (Execute-never). For the EL1&0 translation regime, this bit can restrict execute permissions on writeable pages.

WXN	Meaning
0b0	This control has no effect on memory access permissions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b1	In the EL1&0 translation regime, any region of memory that is writeable at EL1 is XN at EL1, and any region of memory that is writeable at EL0 is XN at EL0.

This bit applies only when SCTLR_EL1.M bit is set.

The WXN bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

nTWE, bit [18]

Traps EL0 execution of WFE instructions to EL1, or to EL2 when HCR_EL2.TGE is 1, from both Execution states, reported using EC syndrome value 0x01.

nTWE	Meaning
0b0	Any attempt to execute a WFE instruction at EL0 is trapped, if the instruction would otherwise have caused the PE to enter a low-power state.
0b1	This control does not cause any instructions to be trapped.

Note

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

BR, bit [17]

When VTCR_EL2.MSA == '0':

Background region enable for EL1 MPU memory regions.

BR	Meaning
0b0	Background region disabled for stage 1 EL1&0 translation regime.
0b1	Background region enabled for stage 1 EL1&0 translation regime.

If the value of HCR_EL2.{DC, TGE} is not {0, 0}, then PE behaves as if the value of the SCTLR_EL1.BR field is 0 for all purposes other than returning the value of a direct read of the field.

If EL1 MPU is enabled, then EL0 access that does not match an EL1 MPU region always results in a Translation fault.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Otherwise:

Reserved, RES0.

nTWI, bit [16]

Traps EL0 execution of WFI instructions to EL1, or to EL2 when HCR_EL2.TGE is 1, from both Execution states, reported using EC syndrome value 0x01.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

nTWI	Meaning
0b0	Any attempt to execute a WFI instruction at EL0 is trapped, if the instruction would otherwise have caused the PE to enter a low-power state.
0b1	This control does not cause any instructions to be trapped.

Note

Since a WFE or WFI can complete at any time, even without a Wakeup event, the traps on WFE of WFI are not guaranteed to be taken, even if the WFE or WFI is executed when there is no Wakeup event. The only guarantee is that if the instruction does not complete in finite time in the absence of a Wakeup event, the trap will be taken.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

UCT, bit [15]

Traps EL0 accesses to the CTR_EL0 to EL1, or to EL2 when HCR_EL2.TGE is 1, from AArch64 state only, reported using EC syndrome value 0x18.

UCT	Meaning
0b0	Accesses to the CTR_EL0 from EL0 using AArch64 are trapped.
0b1	This control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

DZE, bit [14]

Traps EL0 execution of DC ZVA instructions to EL1, or to EL2 when HCR_EL2.TGE is 1, from AArch64 state only, reported using EC syndrome value 0x18.

DZE	Meaning
0b0	Any attempt to execute an instruction that this trap applies to at EL0 using AArch64 is trapped. Reading DCZID_EL0.DZP from EL0 returns 1, indicating that the instructions this trap applies to are not supported.
0b1	This control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EnDB, bit [13]

When FEAT_PAuth is implemented:

Controls enabling of pointer authentication of data addresses, using the APDBKey_EL1 key, at EL1 and at EL0 in the EL1&0 translation regime.

EnDB	Meaning
0b0	Pointer authentication of data addresses at EL1 and at EL0 in the EL1&0 translation regime, using the APDBKey_EL1 key, is not enabled.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EnDB	Meaning
0b1	Pointer authentication of data addresses at EL1 and at EL0 in the EL1&0 translation regime, using the APDBKey_EL1 key, is enabled.

Note

This field controls the behavior of the AddPACDB and AuthDB pseudocode functions. When pointer authentication is enabled, AddPACDB returns a copy of a pointer to which a pointer authentication code has been added, and AuthDB returns an authenticated copy of a pointer. When pointer authentication is disabled, both of these functions are NOP.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

I, bit [12]

Stage 1 instruction access Cacheability control, for accesses at EL0 and EL1:

I	Meaning
0b0	All instruction access to Stage 1 Normal memory from EL0 and EL1 are Stage 1 Non-cacheable. If stage 1 EL1&0 translation is in VMSAv8-64 context and the value of SCTLR_EL1.M is 0, then instruction accesses from stage 1 are to Normal, Outer Shareable, Inner Non-cacheable, Outer Non-cacheable memory. If stage 1 EL1&0 translation is in PMSAv8-64 context and the value of SCTLR_EL1.{BR, M} = {0, 0}, then instruction accesses from stage 1 are to Normal, Outer Shareable, Inner Non-cacheable, Outer Non-cacheable memory.
0b1	This control has no effect on the Stage 1 Cacheability of instruction access to Stage 1 Normal memory from EL0 and EL1. If stage 1 EL1&0 translation is in VMSAv8-64 context and the value of SCTLR_EL1.M is 0, then instruction accesses from stage 1 are to Normal, Outer Shareable, Inner Write-Through, Outer Write-Through memory. If stage 1 EL1&0 translation is in PMSAv8-64 context, and the value of SCTLR_EL1.{BR, M} = {0, 0}, then instruction accesses from stage 1 are to Normal, Outer Shareable, Inner Write-Through, Outer Write-Through memory.

When the value of the HCR_EL2.DC bit is 1, then instruction access to Normal memory from EL0 and EL1 are Cacheable regardless of the value of the SCTLR_EL1.I bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [11]

Reserved, RES1.

EnRCTX, bit [10]

When FEAT_SPECRES is implemented:

Enable EL0 access to the following System instructions:

- CFP RCTX, DVP RCTX and CPP RCTX instructions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EnRCTX	Meaning
0b0	EL0 access to these instructions is disabled, and these instructions are trapped to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1.
0b1	EL0 access to these instructions is enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

UMA, bit [9]

User Mask Access. Traps EL0 execution of MSR and MRS instructions that access the PSTATE.{D, A, I, F} masks to EL1, or to EL2 when HCR_EL2.TGE is 1, from AArch64 state only, reported using EC syndrome value 0x18.

UMA	Meaning
0b0	Any attempt at EL0 using AArch64 to execute an MRS, MSR(register), or MSR(immediate) instruction that accesses the DAIF is trapped.
0b1	This control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [8:7]

Reserved, RES1.

nAA, bit [6]

When FEAT_LSE2 is implemented:

Non-aligned access. This bit controls generation of Alignment faults at EL1 and EL0 under certain conditions.

The following instructions generate an Alignment fault if all bytes being accessed are not within a single naturally aligned 16-byte quantity, for access:

- LDAPR, LDAPRH, LDAPUR, LDAPURH, LDAPURSH, LDAPURSW, LDAR, LDARH, LDLAR, LDLARH.
- STLLR, STLLRH, STLR, STLRH, STLUR, and STLURH.

nAA	Meaning
0b0	Unaligned accesses by the specified instructions generate an Alignment fault.
0b1	This control does not generate Alignment faults.

For a load-acquire instruction that does not have acquire semantics as the result of the destination register, or registers, being ZR, it is IMPLEMENTATION SPECIFIC whether this field behaves as 1 or the programmed value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bit [5]

Reserved, RES0.

SA0, bit [4]

SP Alignment check enable for EL0. When set to 1, if a load or store instruction executed at EL0 uses the SP as the base address and the SP is not aligned to a 16-byte boundary, then an SP alignment fault exception is generated. For more information, see 'SP alignment checking'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SA, bit [3]

SP Alignment check enable. When set to 1, if a load or store instruction executed at EL1 uses the SP as the base address and the SP is not aligned to a 16-byte boundary, then an SP alignment fault exception is generated. For more information, see 'SP alignment checking'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [2]

Stage 1 Cacheability control, for data accesses.

C	Meaning
0b0	All data access to Stage 1 Normal memory from EL0 and EL1, and all Normal memory accesses from unified cache to the EL1&0 Stage 1 translation tables, are treated as Stage 1 Non-cacheable.
0b1	This control has no effect on the Stage 1 Cacheability of: • Data access to Normal memory from EL0 and EL1. • Normal memory accesses to the EL1&0 Stage 1 translation tables.

When the Effective value of the HCR_EL2.DC bit in the current Security state is 1, the PE ignores SCTLR_EL1.C. This means that EL0 and EL1 data accesses to Normal memory are Cacheable.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [1]

Alignment check enable. This is the enable bit for Alignment fault checking at EL1 and EL0.

A	Meaning
0b0	Alignment fault checking is disabled when executing at EL1 or EL0. Alignment checks on some instructions are not disabled by this control. For more information, see 'Alignment of data accesses'.
0b1	Alignment fault checking is enabled when executing at EL1 or EL0. All instructions that load or store one or more registers have an alignment check that the address being accessed is aligned to the size of the data element(s) being accessed. If this check fails it causes an Alignment fault, which is taken as a Data Abort exception.

The reset behavior of this field is:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M, bit [0]

MMU or MPU enable for EL1&0 stage 1 address translation.

This is the enable bit for:

- MPU, if stage 1 EL1&0 translation is in PMSAv8-64 context.
- MMU, if stage 1 EL1&0 translation is in VMSAv8-64 context.

M	Meaning
0b0	EL1 MPU(PMSAv8-64) or MMU(VMSAv8-64) disabled See the SCTLR_EL1.I field for the behavior of instruction accesses to Normal memory.
0b1	EL1 MPU(PMSAv8-64) or MMU(VMSAv8-64) enabled

If the Effective value of HCR_EL2.{DC, TGE} in the current Security state is not {0, 0}, then the PE behaves as if the value of the SCTLR_EL1.M field is 0 for all purposes other than returning the value of a direct read of the field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SCTLR_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, SCTLR_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = SCTLR_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = SCTLR_EL1;
```

`MSR SCTLR_EL1, <Xt>`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        SCTLR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    SCTLR_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.87 SCTLR_EL2, System Control Register (EL2)

The SCTLR_EL2 characteristics are:

Purpose

Provides top-level control of the system, including its memory system, at EL2.

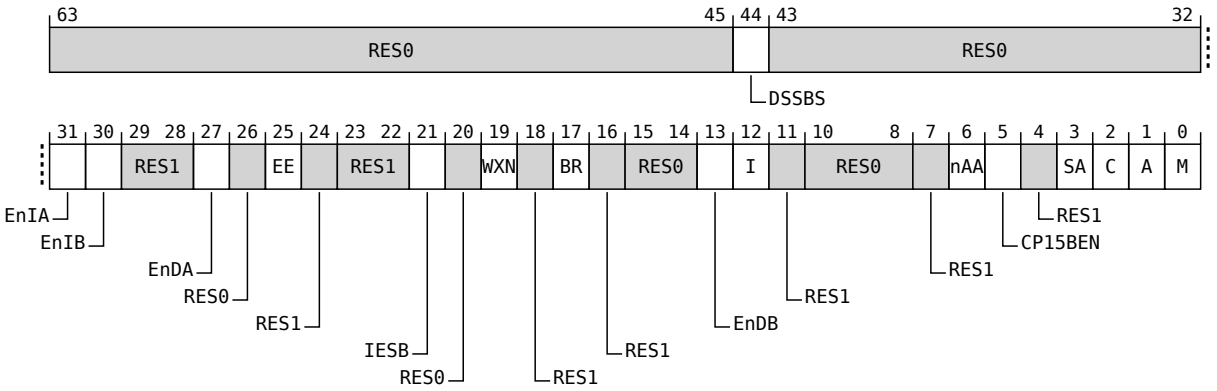
Configuration

There are no configuration notes.

Attributes

SCTLR_EL2 is a 64-bit register.

Field descriptions



Bits [63:45]

Reserved, RES0.

DSSBS, bit [44]

When FEAT_SSBS is implemented:

Default PSTATE.SSBS value on exception entry.

DSSBS	Meaning
0b0	PSTATE.SSBS is set to 0 on an exception to EL2.
0b1	PSTATE.SSBS is set to 1 on an exception to EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

Otherwise:

Reserved, RES0.

Bits [43:32]

Reserved, RES0.

EnIA, bit [31]

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_PAuth is implemented:

Controls enabling of pointer authentication of instruction addresses, using the APIAKey_EL1 key, at EL2 and at EL0 in the EL2&0 translation regime.

EnIA	Meaning
0b0	Pointer authentication of instruction addresses at EL2 and at EL0 in the EL2&0 translation regime, using the APIAKey_EL1 key, is not enabled.
0b1	Pointer authentication of instruction addresses at EL2 and at EL0 in the EL2&0 translation regime, using the APIAKey_EL1 key, is enabled.

Note

This field controls the behavior of the AddPACIA and AuthIA pseudocode functions. When pointer authentication is enabled, AddPACIA returns a copy of a pointer to which a pointer authentication code has been added, and AuthIA returns an authenticated copy of a pointer. When pointer authentication is disabled, both of these functions are NOP.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EnIB, bit [30]

When FEAT_PAuth is implemented:

Controls enabling of pointer authentication of instruction addresses, using the APIBKey_EL1 key, at EL2 and at EL0 in the EL2&0 translation regime.

EnIB	Meaning
0b0	Pointer authentication of instruction addresses at EL2 and at EL0 in the EL2&0 translation regime, using the APIBKey_EL1 key, is not enabled.
0b1	Pointer authentication of instruction addresses at EL2 and at EL0 in the EL2&0 translation regime, using the APIBKey_EL1 key, is enabled.

Note

This field controls the behavior of the AddPACIB and AuthIB pseudocode functions. When pointer authentication is enabled, AddPACIB returns a copy of a pointer to which a pointer authentication code has been added, and AuthIB returns an authenticated copy of a pointer. When pointer authentication is disabled, both of these functions are NOP.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [29:28]

Reserved, RES1.

EnDA, bit [27]

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_PAuth is implemented:

Controls enabling of pointer authentication of data addresses, using the APDAKey_EL1 key, at EL2 and at EL0 in the EL2&0 translation regime.

EnDA	Meaning
0b0	Pointer authentication of data addresses at EL2 and at EL0 in the EL2&0 translation regime, using the APDAKey_EL1 key, is not enabled.
0b1	Pointer authentication of data addresses at EL2 and at EL0 in the EL2&0 translation regime, using the APDAKey_EL1 key, is enabled.

Note

This field controls the behavior of the AddPACDA and AuthDA pseudocode functions. When pointer authentication is enabled, AddPACDA returns a copy of a pointer to which a pointer authentication code has been added, and AuthDA returns an authenticated copy of a pointer. When pointer authentication is disabled, both of these functions are NOP.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [26]

Reserved, RES0.

EE, bit [25]

When FEAT_MixedEnd is implemented:

Endianness of data accesses at EL2.

EE	Meaning
0b0	Explicit data accesses at EL2 are little-endian.
0b1	Explicit data accesses at EL2 are big-endian.

The EE bit is permitted to be cached in a TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an IMPLEMENTATION DEFINED value.

When FEAT_BigEnd is implemented:

Explicit data accesses at EL1, and stage 1 translation table walks in the EL1&0 translation regime are big-endian.

Reserved, RES1.

Otherwise:

Explicit data accesses at EL1, and stage 1 translation table walks in the EL1&0 translation regime are little-endian.

Reserved, RES0.

Bit [24]

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_BigEndEL0 is implemented:

Bit [24]

Explicit data accesses at EL0 are big-endian.

Reserved, RES1.

Otherwise:

Bit [24]

Explicit data accesses at EL0 are little-endian.

Reserved, RES0.

Bits [23:22]

Reserved, RES1.

IESB, bit [21]

When FEAT_IESB is implemented:

Implicit Error Synchronization event enable.

IESB	Meaning
0b0	Disabled.
0b1	An implicit error synchronization event is added: • At each exception taken to EL2. • Before the operational pseudocode of each Exception Return instruction executed at EL2.

When the PE is in Debug state, the effect of this field is CONSTRAINED UNPREDICTABLE, and its Effective value might be 0 or 1 regardless of the value of the field. If the Effective value of the field is 1, then an implicit error synchronization event is added after each DCPSx instruction taken to EL2 and before each DRPS instruction executed at EL2, in addition to the other cases where it is added.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [20]

Reserved, RES0.

WXN, bit [19]

Write permission implies XN (Execute-never). For the EL2 or EL2&0 translation regime, this bit can restrict execute permissions on writeable pages.

WXN	Meaning
0b0	This control has no effect on memory access permissions.
0b1	In the EL2 or EL2&0 translation regime, any region of memory that is writeable at EL2 is XN at EL2. In the EL2&0 translation regime, any region of memory that is writeable at EL0 is XN at EL0.

This bit applies only when SCTLR_EL2.M bit is set.

The WXN bit is permitted to be cached in a TLB.

The reset behavior of this field is:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [18]

Reserved, RES1.

BR, bit [17]

Background region enable for EL2 MPU memory regions.

BR	Meaning
0b0	Background region disabled for stage 1 EL2 translation regime and stage 2 EL1&0 translation regime.
0b1	Background region enabled for stage 1 EL2 translation regime and stage 2 EL1&0 translation regime.

If EL2 MPU is enabled, then EL0 and EL1 access that does not match an EL2 MPU region always results in a Translation fault.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Bit [16]

Reserved, RES1.

Bits [15:14]

Reserved, RES0.

EnDB, bit [13]

When FEAT_PAuth is implemented:

Controls enabling of pointer authentication of data addresses, using the APDBKey_EL1 key, at EL2 and at EL0 in the EL2&0 translation regime.

EnDB	Meaning
0b0	Pointer authentication of data addresses at EL2 and at EL0 in the EL2&0 translation regime, using the APDBKey_EL1 key, is not enabled.
0b1	Pointer authentication of data addresses at EL2 and at EL0 in the EL2&0 translation regime, using the APDBKey_EL1 key, is enabled.

Note

This field controls the behavior of the AddPACDB and AuthDB pseudocode functions. When pointer authentication is enabled, AddPACDB returns a copy of a pointer to which a pointer authentication code has been added, and AuthDB returns an authenticated copy of a pointer. When pointer authentication is disabled, both of these functions are NOP.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

I, bit [12]

Instruction access Cacheability control, for accesses at EL2.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I	Meaning
0b0	All instruction accesses to Normal memory from EL2 are Non-cacheable for all levels of instruction and unified cache. If SCTLR_EL2.{BR, M} == {0, 0}, then instruction accesses from stage 1 of the EL2 translation regime are to Normal, Outer Shareable, Inner Non-cacheable, Outer Non-cacheable memory.
0b1	This control has no effect on the Cacheability of instruction access to Normal memory from EL2. If SCTLR_EL2.{BR, M} = {0, 0}, then instruction accesses from stage 1 of the EL2 translation regime are to Normal, Outer Shareable, Inner Write-Through, Outer Write-Through memory.

This bit has no effect on the EL1&0 translation regime.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Bit [11]

Reserved, RES1.

Bits [10:8]

Reserved, RES0.

Bit [7]

Reserved, RES1.

nAA, bit [6]

When FEAT_LSE2 is implemented:

Non-aligned access. This bit controls generation of Alignment faults under certain conditions at EL2.

The following instructions generate an Alignment fault if all bytes being accessed are not within a single naturally aligned 16-byte quantity, for access:

- LDAPR, LDAPRH, LDAPUR, LDAPURH, LDAPURSH, LDAPURSW, LDAR, LDARH, LDLAR, LDLARH.
- STLLR, STLLRH, STLR, STLRH, STLUR, and STLURH.

nAA	Meaning
0b0	Unaligned accesses by the specified instructions generate an Alignment fault.
0b1	Unaligned accesses by the specified instructions do not generate an Alignment fault.

For a load-acquire instruction that does not have acquire semantics as the result of the destination register, or registers, being ZR, it is IMPLEMENTATION SPECIFIC whether this field behaves as 1 or the programmed value.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

CP15BEN, bit [5]

Access to this field is RES0.

Bit [4]

Reserved, RES1.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SA, bit [3]

SP Alignment check enable. When set to 1, if a load or store instruction executed at EL2 uses the SP as the base address and the SP is not aligned to a 16-byte boundary, then an SP alignment fault exception is generated. For more information, see 'SP alignment checking'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [2]

Data access Cacheability control, for accesses at EL2.

C	Meaning
0b0	All data accesses to Normal memory from EL2 are Non-cacheable for all levels of data and unified cache.
0b1	This control has no effect on the Cacheability of data accesses to Normal memory from EL2.

This bit has no effect on the EL1&0 translation regime.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

A, bit [1]

Alignment check enable. This is the enable bit for Alignment fault checking at EL2.

A	Meaning
0b0	Alignment fault checking is disabled when executing at EL2. Alignment checks on some instructions are not disabled by this control. For more information, see 'Alignment of data accesses'.
0b1	Alignment fault checking is enabled when executing at EL2. All instructions that load or store one or more registers have an alignment check that the address being accessed is aligned to the size of the data element(s) being accessed. If this check fails it causes an Alignment fault, which is taken as a Data Abort exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M, bit [0]

MPU enable for EL2 stage 1 and EL1&0 stage 2 address translation.

M	Meaning
0b0	MPU disabled for EL2 and EL1&0 stage 2 address translation. See the SCTLR_EL2.I field for the behavior of instruction accesses to Normal memory.
0b1	MPU enabled for EL2 and EL1&0 stage 2 address translation.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing SCTLR_EL2

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, SCTLR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = SCTLR_EL2;
```

MSR SCTLR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    SCTLR_EL2 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.88 SCXTNUM_EL0, EL0 Read/Write Software Context Number

The SCXTNUM_EL0 characteristics are:

Purpose

Provides a number that can be used to separate out different context numbers with the EL0 exception level, for the purpose of protecting against side-channels using branch prediction and similar resources.

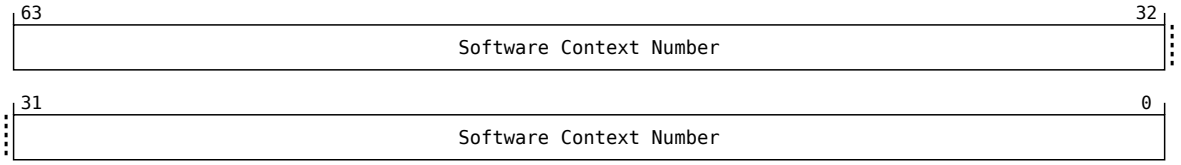
Configuration

This register is present only when FEAT_CSV2_2 is implemented or FEAT_CSV2_1p2 is implemented. Otherwise, direct accesses to SCXTNUM_EL0 are UNDEFINED.

Attributes

SCXTNUM_EL0 is a 64-bit register.

Field descriptions



SCXTNUM, bits [63:0]

Software Context Number. A number to identify the context within the EL0 exception level.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SCXTNUM_EL0

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, SCXTNUM_EL0`

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0000	0b111

```
if !(IsFeatureImplemented(FEAT_CSV2_2) || IsFeatureImplemented(FEAT_CSV2_1p2)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && SCTLR_EL1.TSCXT == '1' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif !ELIsInHost(EL0) && HCR_EL2.EnSCXT == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = SCXTNUM_EL0;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.EnSCXT == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = SCXTNUM_EL0;
elsif PSTATE.EL == EL2 then
    X[t, 64] = SCXTNUM_EL0;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR SCXTNUM_EL0, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0000	0b111

```
if !(IsFeatureImplemented(FEAT_CSV2_2) || IsFeatureImplemented(FEAT_CSV2_1p2)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && SCTLR_EL1.TSCXT == '1' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif !ELIsInHost(EL0) && HCR_EL2.EnSCXT == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        SCXTNUM_EL0 = X[t, 64];
elsif PSTATE.EL == EL1 then
    if HCR_EL2.EnSCXT == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        SCXTNUM_EL0 = X[t, 64];
elsif PSTATE.EL == EL2 then
    SCXTNUM_EL0 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.89 SCXTNUM_EL1, EL1 Read/Write Software Context Number

The SCXTNUM_EL1 characteristics are:

Purpose

Provides a number that can be used to separate out different context numbers with the EL1 exception level, for the purpose of protecting against side-channels using branch prediction and similar resources.

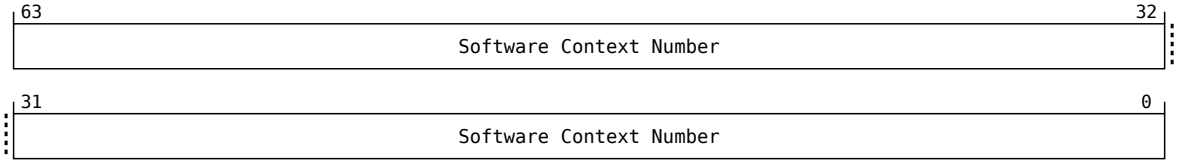
Configuration

This register is present only when FEAT_CSV2_2 is implemented or FEAT_CSV2_1p2 is implemented. Otherwise, direct accesses to SCXTNUM_EL1 are UNDEFINED.

Attributes

SCXTNUM_EL1 is a 64-bit register.

Field descriptions



SCXTNUM, bits [63:0]

Software Context Number. A number to identify the context within the EL1 exception level.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SCXTNUM_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, SCXTNUM_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b1101	0b0000	0b111

```
if !(IsFeatureImplemented(FEAT_CSV2_2) || IsFeatureImplemented(FEAT_CSV2_1p2)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.EnSCXT == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = SCXTNUM_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = SCXTNUM_EL1;
```

`MSR SCXTNUM_EL1, <Xt>`

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b000	0b1101	0b0000	0b111

```
if !(IsFeatureImplemented(FEAT_CSV2_2) || IsFeatureImplemented(FEAT_CSV2_1p2)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.EnSCXT == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        SCXTNUM_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    SCXTNUM_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.90 SCXTNUM_EL2, EL2 Read/Write Software Context Number

The SCXTNUM_EL2 characteristics are:

Purpose

Provides a number that can be used to separate out different context numbers with the EL2 exception level, for the purpose of protecting against side-channels using branch prediction and similar resources.

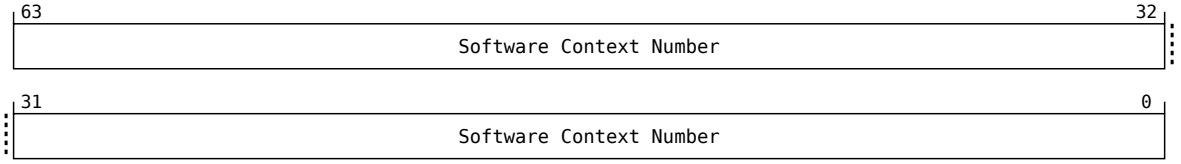
Configuration

This register is present only when FEAT_CSV2_2 is implemented or FEAT_CSV2_1p2 is implemented. Otherwise, direct accesses to SCXTNUM_EL2 are UNDEFINED.

Attributes

SCXTNUM_EL2 is a 64-bit register.

Field descriptions



SCXTNUM, bits [63:0]

Software Context Number. A number to identify the context within the EL2 exception level.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing SCXTNUM_EL2

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, SCXTNUM_EL2`

op0	op1	CRn	CRm	op2
0b11	0b100	0b1101	0b0000	0b111

```
if !(IsFeatureImplemented(FEAT_CSV2_2) || IsFeatureImplemented(FEAT_CSV2_1p2)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = SCXTNUM_EL2;
```

`MSR SCXTNUM_EL2, <Xt>`

op0	op1	CRn	CRm	op2
0b11	0b100	0b1101	0b0000	0b111

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !(IsFeatureImplemented(FEAT_CSV2_2) || IsFeatureImplemented(FEAT_CSV2_1p2)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    SCXTNUM_EL2 = X[t, 64];
```

MRS <Xt>, SCXTNUM_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1101	0b0000	0b111

```
if !(IsFeatureImplemented(FEAT_CSV2_2) || IsFeatureImplemented(FEAT_CSV2_1p2)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.EnSCXT == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = SCXTNUM_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = SCXTNUM_EL1;
```

MSR SCXTNUM_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1101	0b0000	0b111

```
if !(IsFeatureImplemented(FEAT_CSV2_2) || IsFeatureImplemented(FEAT_CSV2_1p2)) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.EnSCXT == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        SCXTNUM_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    SCXTNUM_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.91 TCR_EL1, Translation Control Register (EL1)

The TCR_EL1 characteristics are:

Purpose

The control register for stage 1 of the EL1&0 translation regime.

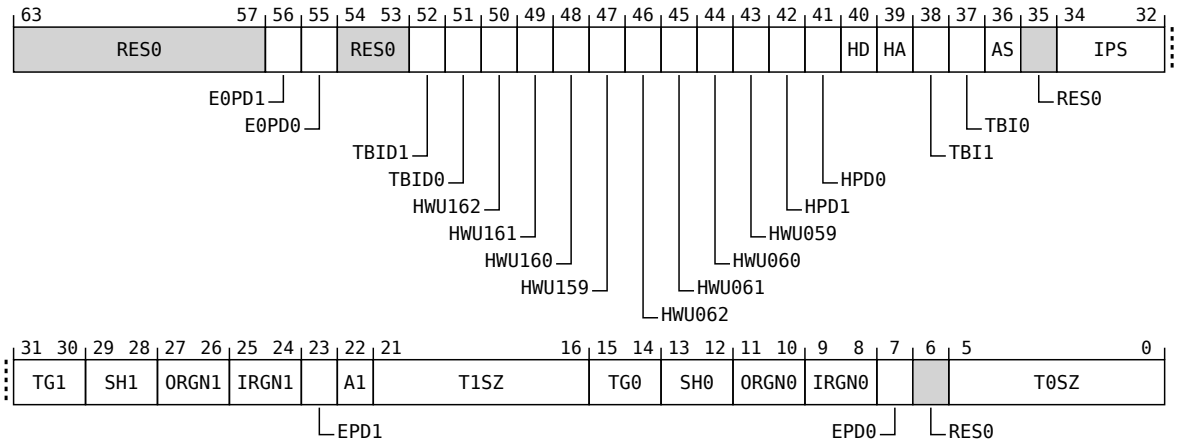
Configuration

There are no configuration notes.

Attributes

TCR_EL1 is a 64-bit register.

Field descriptions



Any of the bits in TCR_EL1, other than the EPDx bits when they have the value 1, and the A1 bit are permitted to be cached in a TLB.

Bits [63:57]

Reserved, RES0.

E0PD1, bit [56]

When VTCR_EL2.MSA == '1' and FEAT_E0PD is implemented:

Faulting control for unprivileged access to any address translated by TTBR1_EL1.

E0PD1	Meaning
0b0	Unprivileged access to any address translated by TTBR1_EL1 will not generate a fault by this mechanism.
0b1	Unprivileged access to any address translated by TTBR1_EL1 will generate a level 0 Translation fault.

Level 0 Translation faults generated as a result of this field are not counted as TLB misses for performance monitoring. The fault should take the same time to generate, whether the address is present in the TLB or not, to mitigate attacks that use fault timing.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

E0PD0, bit [55]

When VTCR_EL2.MSA == '1' and FEAT_E0PD is implemented:

Faulting control for unprivileged access to any address translated by TTBR0_EL1.

E0PD0	Meaning
0b0	Unprivileged access to any address translated by TTBR0_EL1 will not generate a fault by this mechanism.
0b1	Unprivileged access to any address translated by TTBR0_EL1 will generate a level 0 Translation fault.

Level 0 Translation faults generated as a result of this field are not counted as TLB misses for performance monitoring. The fault should take the same time to generate, whether the address is present in the TLB or not, to mitigate attacks that use fault timing.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [54:53]

Reserved, RES0.

TBID1, bit [52]

When VTCR_EL2.MSA == '1' and FEAT_PAuth is implemented:

Controls the use of the top byte of instruction addresses for address matching.

For the purpose of this field, all cache maintenance and address translation instructions that perform address translation are treated as data accesses.

For more information, see 'Address tagging'.

TBID1	Meaning
0b0	TCR_EL1.TBI1 applies to Instruction and Data accesses.
0b1	TCR_EL1.TBI1 applies to Data accesses only.

This affects addresses where the address would be translated by tables pointed to by TTBR1_EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TBID0, bit [51]

When VTCR_EL2.MSA == '1' and FEAT_PAuth is implemented:

Controls the use of the top byte of instruction addresses for address matching.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For the purpose of this field, all cache maintenance and address translation instructions that perform address translation are treated as data accesses.

For more information, see 'Address tagging'.

TBID0	Meaning
0b0	TCR_EL1.TBI0 applies to Instruction and Data accesses.
0b1	TCR_EL1.TBI0 applies to Data accesses only.

This affects addresses where the address would be translated by tables pointed to by TTBR0_EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When FEAT_PAuth is implemented and VTCR_EL2.MSA == '0':

Controls the use of the top byte of instruction addresses for address matching.

For the purpose of this field, all cache maintenance and address translation instructions that perform address translation are treated as data accesses.

For more information, see 'Address tagging'.

TBID0	Meaning
0b0	TCR_EL1.TBI0 applies to Instruction and Data accesses.
0b1	TCR_EL1.TBI0 applies to Data accesses only.

This affects addresses where the address would be translated by EL1 MPU.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU162, bit [50]

When VTCR_EL2.MSA == '1' and FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[62] of the stage 1 translation table Block or Page entry for translations using TTBR1_EL1.

HWU162	Meaning
0b0	For translations using TTBR1_EL1, bit[62] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1	For translations using TTBR1_EL1, bit[62] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL1.HPD1 is 1.

The Effective value of this field is 0 if the value of TCR_EL1.HPD1 is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

HWU161, bit [49]

When VTCR_EL2.MSA == '1' and FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[61] of the stage 1 translation table Block or Page entry for translations using TTBR1_EL1.

HWU161	Meaning
0b0	For translations using TTBR1_EL1, bit[61] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1	For translations using TTBR1_EL1, bit[61] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL1.HPD1 is 1.

The Effective value of this field is 0 if the value of TCR_EL1.HPD1 is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU160, bit [48]

When VTCR_EL2.MSA == '1' and FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[60] of the stage 1 translation table Block or Page entry for translations using TTBR1_EL1.

HWU160	Meaning
0b0	For translations using TTBR1_EL1, bit[60] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1	For translations using TTBR1_EL1, bit[60] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL1.HPD1 is 1.

The Effective value of this field is 0 if the value of TCR_EL1.HPD1 is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU159, bit [47]

When VTCR_EL2.MSA == '1' and FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[59] of the stage 1 translation table Block or Page entry for translations using TTBR1_EL1.

HWU159	Meaning
0b0	For translations using TTBR1_EL1, bit[59] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

HWU159	Meaning
0b1	For translations using TTBR1_EL1, bit[59] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL1.HPD1 is 1.

The Effective value of this field is 0 if the value of TCR_EL1.HPD1 is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU062, bit [46]

When VTCR_EL2.MSA == '1' and FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[62] of the stage 1 translation table Block or Page entry for translations using TTBR0_EL1.

HWU062	Meaning
0b0	For translations using TTBR0_EL1, bit[62] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1	For translations using TTBR0_EL1, bit[62] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL1.HPD0 is 1.

The Effective value of this field is 0 if the value of TCR_EL1.HPD0 is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU061, bit [45]

When VTCR_EL2.MSA == '1' and FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[61] of the stage 1 translation table Block or Page entry for translations using TTBR0_EL1.

HWU061	Meaning
0b0	For translations using TTBR0_EL1, bit[61] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1	For translations using TTBR0_EL1, bit[61] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL1.HPD0 is 1.

The Effective value of this field is 0 if the value of TCR_EL1.HPD0 is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

HWU060, bit [44]

When VTCR_EL2.MSA == '1' and FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[60] of the stage 1 translation table Block or Page entry for translations using TTBR0_EL1.

HWU060	Meaning
0b0	For translations using TTBR0_EL1, bit[60] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1	For translations using TTBR0_EL1, bit[60] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL1.HPD0 is 1.

The Effective value of this field is 0 if the value of TCR_EL1.HPD0 is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HWU059, bit [43]

When VTCR_EL2.MSA == '1' and FEAT_HPDS2 is implemented:

Hardware Use. Indicates IMPLEMENTATION DEFINED hardware use of bit[59] of the stage 1 translation table Block or Page entry for translations using TTBR0_EL1.

HWU059	Meaning
0b0	For translations using TTBR0_EL1, bit[59] of each stage 1 translation table Block or Page entry cannot be used by hardware for an IMPLEMENTATION DEFINED purpose.
0b1	For translations using TTBR0_EL1, bit[59] of each stage 1 translation table Block or Page entry can be used by hardware for an IMPLEMENTATION DEFINED purpose if the value of TCR_EL1.HPD0 is 1.

The Effective value of this field is 0 if the value of TCR_EL1.HPD0 is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HPD1, bit [42]

When VTCR_EL2.MSA == '1' and FEAT_HPDS is implemented:

Hierarchical Permission Disables. This affects the hierarchical control bits, APTable, PXNTable, and UXNTable, except NSTable, in the translation tables pointed to by TTBR1_EL1.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

HPD1	Meaning
0b0	Hierarchical permissions are enabled.
0b1	Hierarchical permissions are disabled.

When disabled, the permissions are treated as if the bits are zero.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HPD0, bit [41]

When VTCR_EL2.MSA == '1' and FEAT_HPDS is implemented:

Hierarchical Permission Disables. This affects the hierarchical control bits, APTable, PXNTable, and UXNTable, except NSTable, in the translation tables pointed to by TTBR0_EL1.

HPD0	Meaning
0b0	Hierarchical permissions are enabled.
0b1	Hierarchical permissions are disabled.

When disabled, the permissions are treated as if the bits are zero.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HD, bit [40]

When VTCR_EL2.MSA == '1' and FEAT_HAFDBS is implemented:

Hardware management of dirty state in stage 1 translations from EL0 and EL1.

HD	Meaning
0b0	Stage 1 hardware management of dirty state disabled.
0b1	Stage 1 hardware management of dirty state enabled.

When the Effective value of TCR_EL1.HA is 0, this field behaves as 0 for all purposes other than a direct read of the value of this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HA, bit [39]

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When VTCR_EL2.MSA == '1' and FEAT_HAFDBS is implemented:

Hardware Access flag update in stage 1 translations from EL0 and EL1.

HA	Meaning
0b0	Stage 1 Access flag update disabled.
0b1	Stage 1 Access flag update enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When VTCR_EL2.MSA == '0':

Reserved, RES0.

Otherwise:

Reserved, RES0.

TBI1, bit [38]

When VTCR_EL2.MSA == '1':

Top Byte ignored. Indicates whether the top byte of an address is used for address match for the TTBR1_EL1 region, or ignored and used for tagged addresses.

TBI1	Meaning
0b0	Top Byte used in the address calculation.
0b1	Top Byte ignored in the address calculation.

This affects addresses generated in EL0 and EL1 using AArch64 where the address would be translated by tables pointed to by TTBR1_EL1. It has an effect whether the EL1&0 translation regime is enabled or not.

If FEAT_PAuth is implemented and TCR_EL1.TBID1 is 1, then this field only applies to Data accesses.

Otherwise, if the value of TBI1 is 1 and bit [55] of the target address to be stored to the PC is 1, then bits[63:56] of that target address are also set to 1 before the address is stored in the PC, in the following cases:

- A branch or procedure return within EL0 or EL1.
- An exception taken to EL1.
- An exception return to EL0 or EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When VTCR_EL2.MSA == '0':

Reserved, RES0.

Otherwise:

Reserved, RES0.

TBI0, bit [37]

When VTCR_EL2.MSA == '1':

Top Byte ignored. Indicates whether the top byte of an address is used for address match for the TTBR0_EL1 region, or ignored and used for tagged addresses.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TBI0	Meaning
0b0	Top Byte used in the address calculation.
0b1	Top Byte ignored in the address calculation.

This affects addresses generated in EL0 and EL1 using AArch64 where the address would be translated by tables pointed to by TTBR0_EL1. It has an effect whether the EL1&0 translation regime is enabled or not.

If FEAT_PAuth is implemented and TCR_EL1.TBID0 is 1, then this field only applies to Data accesses.

Otherwise, if the value of TBI0 is 1 and bit [55] of the target address to be stored to the PC is 0, then bits[63:56] of that target address are also set to 0 before the address is stored in the PC, in the following cases:

- A branch or procedure return within EL0 or EL1.
- An exception taken to EL1.
- An exception return to EL0 or EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When VTCR_EL2.MSA == '0':

Top Byte ignored. Indicates whether the top byte of an address is used for an address match for the EL1 MPU regions, or ignored and used for tagged addresses.

TBI0	Meaning
0b0	Top Byte used in the address calculation.
0b1	Top Byte ignored in the address calculation.

This affects addresses generated in EL0 and EL1, where the address would be translated by the EL1 MPU.

If FEAT_PAuth is implemented and TCR_EL1.TBID0 is 1, then this field only applies to Data accesses.

Otherwise, if the value of TBI0 is 1 and bit [55] of the target address to be stored to the PC is 0, then bits[63:56] of that target address are also set to 0 before the address is stored in the PC, in the following cases:

- A branch or procedure return within EL0 or EL1.
- An exception taken to EL1.
- An exception return to EL0 or EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

AS, bit [36]

When VTCR_EL2.MSA == '1':

ASID Size.

AS	Meaning
0b0	8 bit - the upper 8 bits of TTBR0_EL1 and TTBR1_EL1 are ignored by hardware for every purpose except reading back the register, and are treated as if they are all zeros for when used for allocation and matching entries in the TLB.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

AS	Meaning
0b1	16 bit - the upper 16 bits of TTBR0_EL1 and TTBR1_EL1 are used for allocation and matching in the TLB.

If the implementation has only 8 bits of ASID, this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When VTCR_EL2.MSA == '0':

ASID Size.

AS	Meaning
0b0	8 bit - the upper 8 bits of TTBR0_EL1 are ignored by hardware for every purpose except reading back the register, and are treated as if they are all zeros when used for address matching.
0b1	16 bit - the upper 16 bits of TTBR0_EL1 are used for address matching.

If the implementation has only 8 bits of ASID, this field is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [35]

Reserved, RES0.

IPS, bits [34:32]

When VTCR_EL2.MSA == '1':

Intermediate Physical Address Size.

IPS	Meaning
0b000	32 bits, 4GB.
0b001	36 bits, 64GB.
0b010	40 bits, 1TB.
0b011	42 bits, 4TB.
0b100	44 bits, 16TB.
0b101	48 bits, 256TB.
0b110	52 bits, 4PB.
0b111	56 bits, 64PB.

All other values are reserved.

The reserved values behave in the same way as the 0b101 or 0b110 encoding, but software must not rely on this property as the behavior of the reserved values might change in a future revision of the architecture.

The values 0b110 and 0b111 represent different output address sizes depending on implementation choices and translation configuration.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The following table captures the output address size represented by the value 0b110:

Descriptor Format	ID_AA64MMFR0_EL1.PARange	Translation Granule	DS[1] | Represented OA size |
Any	0b0101	Any	Any	48 bits, 256TB
VMSAv8-64	0b011x	4KB or 16KB	0	48 bits, 256TB
VMSAv8-64	0b011x	4KB or 16KB	1	52 bits, 4PB
VMSAv8-64	0b011x	64KB	N/A	52 bits, 4PB
VMSAv9-128	0b011x	Any	N/A	52 bits, 4PB

[1] This column represents the value of TCR_EL1.DS.

The following table captures the output address size represented by the value 0b111:

Descriptor Format	ID_AA64MMFR0_EL1.PARange	Translation Granule	Represented OA size
Any	0b0110	Any	OA size represented by 0b110
VMSAv8-64	0b0111	Any	OA size represented by 0b110
VMSAv9-128	0b0111	Any	56 bits, 64PB

If 52-bit PA is supported, and the translation table descriptors cannot express an OA larger than 48-bits, then bits[51:48] of every translation table base address are treated as 0b0000 for the stage of translation controlled by TCR_EL1.

If 56-bit PA is supported, and the translation table descriptors cannot express an OA larger than 52-bits, then bits[55:52] of every translation table base address are treated as 0b0000 for the stage of translation controlled by TCR_EL1.

If the output address size represented by this field is larger than the supported PA size expressed in ID_AA64MMFR0_EL1.PARange, then the output address size is treated as being the same as the supported PA size. Arm strongly recommends that software avoids configuring this field to a value representing an output address size larger than the supported PA size.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TG1, bits [31:30]

When VTCR_EL2.MSA == '1':

Granule size for the TTBR1_EL1.

TG1	Meaning
0b01	16KB.
0b10	4KB.
0b11	64KB.

Other values are reserved.

If the value is programmed to either a reserved value or a size that has not been implemented, then the hardware will treat the field as if it has been programmed to an IMPLEMENTATION DEFINED choice of the sizes that has been implemented for all purposes other than the value read back from this register.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

It is IMPLEMENTATION DEFINED whether the value read back is the value programmed or the value that corresponds to the size chosen.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SH1, bits [29:28]

When VTCR_EL2.MSA == '1':

Shareability attribute for memory associated with translation table walks using TTBR1_EL1.

SH1	Meaning
0b00	Non-shareable.
0b10	Outer Shareable.
0b11	Inner Shareable.

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

ORGN1, bits [27:26]

When VTCR_EL2.MSA == '1':

Outer cacheability attribute for memory associated with translation table walks using TTBR1_EL1.

ORGN1	Meaning
0b00	Normal memory, Outer Non-cacheable.
0b01	Normal memory, Outer Write-Back Read-Allocate Write-Allocate Cacheable.
0b10	Normal memory, Outer Write-Through Read-Allocate No Write-Allocate Cacheable.
0b11	Normal memory, Outer Write-Back Read-Allocate No Write-Allocate Cacheable.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

IRGN1, bits [25:24]

When VTCR_EL2.MSA == '1':

Inner cacheability attribute for memory associated with translation table walks using TTBR1_EL1.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IRGN1	Meaning
0b00	Normal memory, Inner Non-cacheable.
0b01	Normal memory, Inner Write-Back Read-Allocate Write-Allocate Cacheable.
0b10	Normal memory, Inner Write-Through Read-Allocate No Write-Allocate Cacheable.
0b11	Normal memory, Inner Write-Back Read-Allocate No Write-Allocate Cacheable.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EPD1, bit [23]

When VTCR_EL2.MSA == '1':

Translation table walk disable for translations using TTBR1_EL1. This bit controls whether a translation table walk is performed on a TLB miss, for an address that is translated using TTBR1_EL1. The encoding of this bit is:

EPD1	Meaning
0b0	Perform translation table walks using TTBR1_EL1.
0b1	A TLB miss on an address that is translated using TTBR1_EL1 generates a Translation fault. No translation table walk is performed.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

A1, bit [22]

When VTCR_EL2.MSA == '1':

Selects whether TTBR0_EL1 or TTBR1_EL1 defines the ASID. The encoding of this bit is:

A1	Meaning
0b0	TTBR0_EL1.ASID defines the ASID.
0b1	TTBR1_EL1.ASID defines the ASID.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

T1SZ, bits [21:16]

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When VTCR_EL2.MSA == '1':

The size offset of the memory region addressed by TTBR1_EL1. The region size is $2^{(64-T1SZ)}$ bytes.

The maximum and minimum possible values for T1SZ depend on the level of translation table and the memory translation granule size, as described in the AArch64 Virtual Memory System Architecture chapter.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TG0, bits [15:14]

When VTCR_EL2.MSA == '1':

Granule size for the TTBR0_EL1.

TG0	Meaning
0b00	4KB
0b01	64KB
0b10	16KB

Other values are reserved.

If the value is programmed to either a reserved value or a size that has not been implemented, then the hardware will treat the field as if it has been programmed to an IMPLEMENTATION DEFINED choice of the sizes that has been implemented for all purposes other than the value read back from this register.

It is IMPLEMENTATION DEFINED whether the value read back is the value programmed or the value that corresponds to the size chosen.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SH0, bits [13:12]

When VTCR_EL2.MSA == '1':

Shareability attribute for memory associated with translation table walks using TTBR0_EL1.

SH0	Meaning
0b00	Non-shareable
0b10	Outer Shareable
0b11	Inner Shareable

Other values are reserved. The effect of programming this field to a Reserved value is that behavior is CONSTRAINED UNPREDICTABLE.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

ORGN0, bits [11:10]

When VTCR_EL2.MSA == '1':

Outer cacheability attribute for memory associated with translation table walks using TTBR0_EL1.

ORGN0	Meaning
0b00	Normal memory, Outer Non-cacheable.
0b01	Normal memory, Outer Write-Back Read-Allocate Write-Allocate Cacheable.
0b10	Normal memory, Outer Write-Through Read-Allocate No Write-Allocate Cacheable.
0b11	Normal memory, Outer Write-Back Read-Allocate No Write-Allocate Cacheable.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

IRGN0, bits [9:8]

When VTCR_EL2.MSA == '1':

Inner cacheability attribute for memory associated with translation table walks using TTBR0_EL1.

IRGN0	Meaning
0b00	Normal memory, Inner Non-cacheable.
0b01	Normal memory, Inner Write-Back Read-Allocate Write-Allocate Cacheable.
0b10	Normal memory, Inner Write-Through Read-Allocate No Write-Allocate Cacheable.
0b11	Normal memory, Inner Write-Back Read-Allocate No Write-Allocate Cacheable.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EPD0, bit [7]

When VTCR_EL2.MSA == '1':

Translation table walk disable for translations using TTBR0_EL1. This bit controls whether a translation table walk is performed on a TLB miss, for an address that is translated using TTBR0_EL1. The encoding of this bit is:

EPD0	Meaning
0b0	Perform translation table walks using TTBR0_EL1.
0b1	A TLB miss on an address that is translated using TTBR0_EL1 generates a Translation fault. No translation table walk is performed.

The reset behavior of this field is:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [6]

Reserved, RES0.

T0SZ, bits [5:0]

When VTCR_EL2.MSA == '1':

The size offset of the memory region addressed by TTBR0_EL1. The region size is $2^{(64-T0SZ)}$ bytes.

The maximum and minimum possible values for T0SZ depend on the level of translation table and the memory translation granule size, as described in the AArch64 Virtual Memory System Architecture chapter.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Accessing TCR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, TCR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = TCR_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = TCR_EL1;
```

```
MSR TCR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        TCR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    TCR_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.92 TCR_EL2, Translation Control Register (EL2)

The TCR_EL2 characteristics are:

Purpose

This register controls stage 1 of the EL2 translation regime, translated by the EL2-MPU.

Configuration

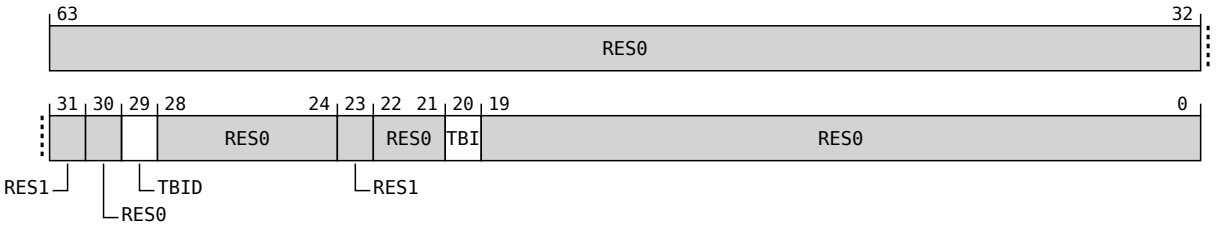
There are no configuration notes.

Attributes

TCR_EL2 is a 64-bit register.

Field descriptions

IsProfile(PROFILE_R)



Any of the bits in TCR_EL2 are permitted to be cached in a TLB.

Bits [63:32]

Reserved, RES0.

Bit [31]

Reserved, RES1.

Bit [30]

Reserved, RES0.

TBID, bit [29]

When FEAT_PAuth is implemented:

Controls the use of the top byte of instruction addresses for address matching.

For the purpose of this field, all cache maintenance and address translation instructions that perform address translation are treated as data accesses.

For more information, see 'Address tagging'.

TBID	Meaning
0b0	TCR_EL2.TBI applies to Instruction and Data accesses.
0b1	TCR_EL2.TBI applies to Data accesses only.

This affects addresses where the address would be translated by EL2 MPU.

The reset behavior of this field is:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [28:24]

Reserved, RES0.

Bit [23]

Reserved, RES1.

Bits [22:21]

Reserved, RES0.

TBI, bit [20]

Top Byte ignored. Indicates whether the top byte of an address is used for address match for the EL2 MPU regions, or ignored and used for tagged addresses.

For more information, see 'Address tagging'.

TBI	Meaning
0b0	Top Byte used in the address calculation.
0b1	Top Byte ignored in the address calculation.

This affects addresses generated in EL2 using AArch64 where the address would be translated by the EL2 MPU. It has an effect whether the EL2 translation regime is enabled or not.

If FEAT_PAuth is implemented and TCR_EL2.TBID is 1, then this field only applies to Data accesses.

If the value of TBI is 1, then bits[63:56] of that target address are also set to 0 before the address is stored in the PC, in the following cases:

- A branch or procedure return within EL2.
- An exception taken to EL2.
- An exception return to EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [19:0]

Reserved, RES0.

Accessing TCR_EL2

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, TCR_EL2`

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = TCR_EL2;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR TCR_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    TCR_EL2 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.93 TPIDR_EL0, EL0 Read/Write Software Thread ID Register

The TPIDR_EL0 characteristics are:

Purpose

Provides a location where software executing at EL0 can store thread identifying information, for OS management purposes.

The PE makes no use of this register.

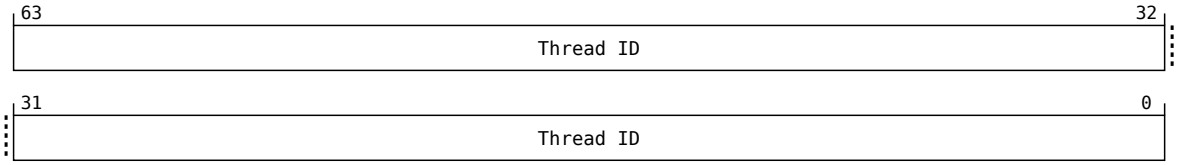
Configuration

There are no configuration notes.

Attributes

TPIDR_EL0 is a 64-bit register.

Field descriptions



ThreadID, bits [63:0]

Thread ID. Thread identifying information stored by software running at this Exception level.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing TPIDR_EL0

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, TPIDR_EL0`

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0000	0b010

```
if PSTATE.EL == EL0 then
    X[t, 64] = TPIDR_EL0;
elsif PSTATE.EL == EL1 then
    X[t, 64] = TPIDR_EL0;
elsif PSTATE.EL == EL2 then
    X[t, 64] = TPIDR_EL0;
```

`MSR TPIDR_EL0, <Xt>`

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0000	0b010

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if PSTATE.EL == EL0 then
    TPIDR_EL0 = X[t, 64];
elsif PSTATE.EL == EL1 then
    TPIDR_EL0 = X[t, 64];
elsif PSTATE.EL == EL2 then
    TPIDR_EL0 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.94 TPIDR_EL1, EL1 Software Thread ID Register

The TPIDR_EL1 characteristics are:

Purpose

Provides a location where software executing at EL1 can store thread identifying information, for OS management purposes.

The PE makes no use of this register.

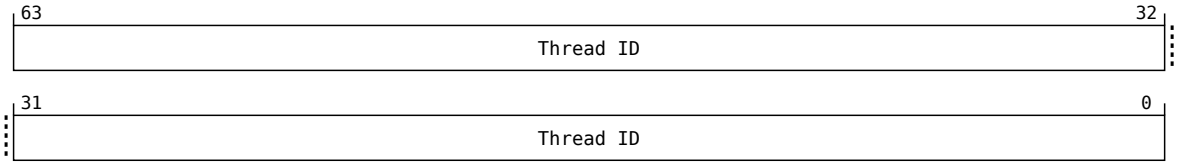
Configuration

There are no configuration notes.

Attributes

TPIDR_EL1 is a 64-bit register.

Field descriptions



ThreadID, bits [63:0]

Thread ID. Thread identifying information stored by software running at this Exception level.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing TPIDR_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, TPIDR_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b1101	0b0000	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    X[t, 64] = TPIDR_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = TPIDR_EL1;
```

`MSR TPIDR_EL1, <Xt>`

op0	op1	CRn	CRm	op2
0b11	0b000	0b1101	0b0000	0b100

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    TPIDR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    TPIDR_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.95 TPIDR_EL2, EL2 Software Thread ID Register

The TPIDR_EL2 characteristics are:

Purpose

Provides a location where software executing at EL2 can store thread identifying information, for OS management purposes.

The PE makes no use of this register.

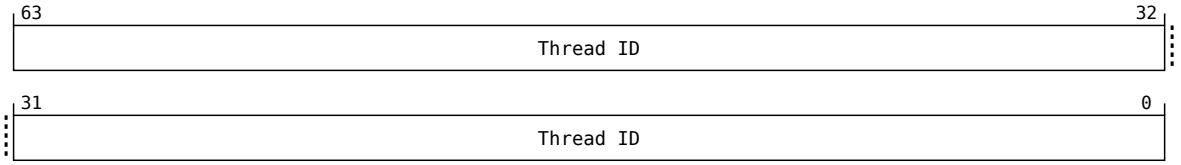
Configuration

There are no configuration notes.

Attributes

TPIDR_EL2 is a 64-bit register.

Field descriptions



ThreadID, bits [63:0]

Thread ID. Thread identifying information stored by software running at this Exception level.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing TPIDR_EL2

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, TPIDR_EL2`

op0	op1	CRn	CRm	op2
0b11	0b100	0b1101	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = TPIDR_EL2;
```

`MSR TPIDR_EL2, <Xt>`

op0	op1	CRn	CRm	op2
0b11	0b100	0b1101	0b0000	0b010

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    TPIDR_EL2 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.96 TPIDRRO_EL0, EL0 Read-Only Software Thread ID Register

The TPIDRRO_EL0 characteristics are:

Purpose

Provides a location where software executing at EL1 or higher can store thread identifying information that is visible to software executing at EL0, for OS management purposes.

The PE makes no use of this register.

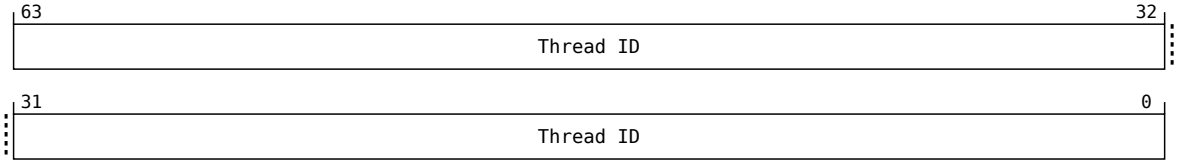
Configuration

There are no configuration notes.

Attributes

TPIDRRO_EL0 is a 64-bit register.

Field descriptions



ThreadID, bits [63:0]

Thread ID. Thread identifying information stored by software running at this Exception level.

Accessing TPIDRRO_EL0

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, TPIDRRO_EL0`

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0000	0b011

```
if PSTATE.EL == EL0 then
    X[t, 64] = TPIDRRO_EL0;
elsif PSTATE.EL == EL1 then
    X[t, 64] = TPIDRRO_EL0;
elsif PSTATE.EL == EL2 then
    X[t, 64] = TPIDRRO_EL0;
```

`MSR TPIDRRO_EL0, <Xt>`

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0000	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    TPIDRRO_EL0 = X[t, 64];
elsif PSTATE.EL == EL2 then
    TPIDRRO_EL0 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.97 TTBR0_EL1, Translation Table Base Register 0 (EL1)

The TTBR0_EL1 characteristics are:

Purpose

Holds the base address of the translation table for the initial lookup for stage 1 of the translation of an address from the lower VA range in the EL1&0 translation regime, and other information for this translation regime.

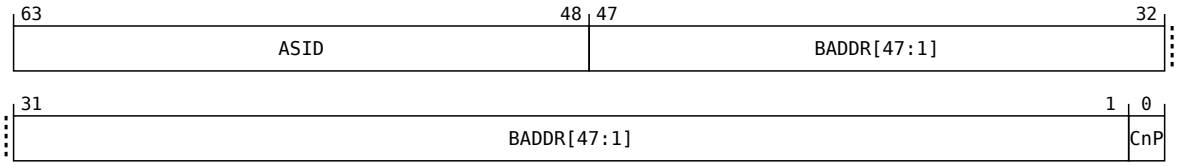
Configuration

There are no configuration notes.

Attributes

TTBR0_EL1 is a 64-bit register.

Field descriptions



ASID, bits [63:48]

When VTCR_EL2.MSA == '1':

An ASID for the translation table base address. The TCR_EL1.A1 field selects either TTBR0_EL1.ASID or TTBR1_EL1.ASID.

If the implementation has only 8 bits of ASID, then the upper 8 bits of this field are RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When VTCR_EL2.MSA == '0':

An ASID for addresses defined by the current EL1 MPU configuration.

If the implementation has only 8 bits of ASID, then the upper 8 bits of this field are RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

BADDR[47:1], bits [47:1]

When VTCR_EL2.MSA == '1':

Translation table base address:

- Bits A[47:x] of the stage 1 translation table base address bits are in register bits[47:x].
- Bits A[(x-1):0] of the stage 1 translation table base address are zero.

Address bit x is the minimum address bit required to align the translation table to the size of the table. The AArch64 Virtual Memory System Architecture chapter describes how x is calculated based on the value of TCR_EL1.T0SZ, the translation stage, and the translation granule size.

The BADDR field represents a 52-bit address if any of the following apply:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The value of TCR_EL1.IPS represents an OA size of 52 bits.
- The Effective value of TCR_EL1.DS is 1.

When TTBR0_EL1.BADDR represents a 52-bit addresses, all of the following apply:

- Bits A[51:48] of the stage 1 translation table base address bits are in register bits[5:2].
- Register bit[1] is RES0.
- The smallest permitted value of x is 6.
- When x>6, register bits[(x-1):6] are RES0.

Otherwise, all of the following apply:

- Register bits[(x-1):1] are RES0.
- If 52-bit PA is supported, then bits A[51:48] of the stage 1 translation table base address are treated as `0b0000`.

Note

If BADDR represents a 52-bit address, and the translation table has fewer than eight entries, the table must be aligned to 64 bytes. Otherwise the translation table must be aligned to the size of the table.
For the 64KB granule, if 52-bit PA is not supported, and the value of TCR_EL1.IPS is `0b110` or `0b111`, one of the following IMPLEMENTATION DEFINED behaviors occur:

- BADDR uses the extended format to represent a 52-bit base address.
- BADDR does not use the extended format.

If any register bit[47:1] that is defined as RES0 has the value 1 when a translation table walk is done using TTBR0_EL1, then the translation table base address might be misaligned, with effects that are CONSTRAINED UNPREDICTABLE, and must be one of the following:

- Bits A[(x-1):0] of the stage 1 translation table base address are treated as if all the bits are zero. The value read back from the corresponding register bits is either the value written to the register or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

CnP, bit [0]

When VTCR_EL2.MSA == '1' and FEAT_TTCNP is implemented:

Common not Private. This bit indicates whether each entry that is pointed to by TTBR0_EL1 is a member of a common set that can be used by every PE in the Inner Shareable domain for which the value of TTBR0_EL1.CnP is 1.

CnP	Meaning
`0b0`	The translation table entries pointed to by TTBR0_EL1, for the current translation regime and ASID, are permitted to differ from corresponding entries for TTBR0_EL1 for other PEs in the Inner Shareable domain. This is not affected by: • The value of TTBR0_EL1.CnP on those other PEs. • The value of the current ASID. • If EL2 is implemented and enabled in the current Security state, the value of the current VMID.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CnP	Meaning
0b1	The translation table entries pointed to by TTBR0_EL1 are the same as the translation table entries for every other PE in the Inner Shareable domain for which the value of TTBR0_EL1.CnP is 1 and all of the following apply: • The translation table entries are pointed to by TTBR0_EL1. • The translation tables relate to the same translation regime. • The ASID is the same as the current ASID. • If EL2 is implemented and enabled in the current Security state, the value of the current VMID.

This bit is permitted to be cached in a TLB.

When a TLB combines entries from stage 1 translation and stage 2 translation into a single entry, that entry can only be shared between different PEs if the value of the CnP bit is 1 for both stage 1 and stage 2.

Note

If the value of the TTBR0_EL1.CnP bit is 1 on multiple PEs in the same Inner Shareable domain and those TTBR0_EL1s do not point to the same translation table entries when the other conditions specified for the case when the value of CnP is 1 apply, then the results of translations are CONSTRAINED UNPREDICTABLE, see 'CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When VTCR_EL2.MSA == '0':

Reserved, RES0.

Otherwise:

Reserved, RES0.

Accessing TTBR0_EL1

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, TTBR0_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = TTBR0_EL1<63:0>;
elsif PSTATE.EL == EL2 then
    X[t, 64] = TTBR0_EL1<63:0>;
```

MSR TTBR0_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0000	0b000

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        TTBR0_EL1<63:0> = X[t, 64];
elsif PSTATE.EL == EL2 then
    TTBR0_EL1<63:0> = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.98 TTBR1_EL1, Translation Table Base Register 1 (EL1)

The TTBR1_EL1 characteristics are:

Purpose

Holds the base address of the translation table for the initial lookup for stage 1 of the translation of an address from the higher VA range in the EL1&0 stage 1 translation regime, and other information for this translation regime.

Configuration

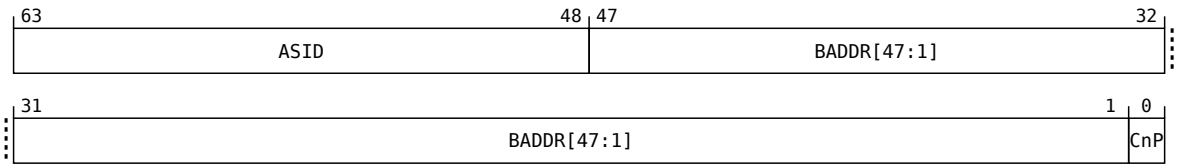
In a PMSAv8-64 only implementation, this register is UNDEFINED.

This register is present only when architecture implements VMSA extension. Otherwise, direct accesses to TTBR1_EL1 are UNDEFINED.

Attributes

TTBR1_EL1 is a 64-bit register.

Field descriptions



ASID, bits [63:48]

When VTCR_EL2.MSA == '1':

An ASID for the translation table base address. The TCR_EL1.A1 field selects either TTBR0_EL1.ASID or TTBR1_EL1.ASID.

If the implementation has only 8 bits of ASID, then the upper 8 bits of this field are RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

BADDR[47:1], bits [47:1]

When VTCR_EL2.MSA == '1':

Translation table base address:

- Bits A[47:x] of the stage 1 translation table base address bits are in register bits[47:x].
- Bits A[(x-1):0] of the stage 1 translation table base address are zero.

Address bit x is the minimum address bit required to align the translation table to the size of the table. The AArch64 Virtual Memory System Architecture chapter describes how x is calculated based on the value of TCR_EL1.T1SZ, the translation stage, and the translation granule size.

The BADDR field represents a 52-bit address if any of the following apply:

- The value of TCR_EL1.IPS represents an OA size of 52 bits.
- The Effective value of TCR_EL1.DS is 1.

When TTBR1_EL1.BADDR represents a 52-bit addresses, all of the following apply:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Bits A[51:48] of the stage 1 translation table base address bits are in register bits[5:2].
- Register bit[1] is RES0.
- The smallest permitted value of x is 6.
- When x>6, register bits[(x-1):6] are RES0.

Otherwise, all of the following apply:

- Register bits[(x-1):1] are RES0.
- If 52-bit PA is supported, then bits A[51:48] of the stage 1 translation table base address are treated as `0b0000`.

Note

If BADDR represents a 52-bit address, and the translation table has fewer than eight entries, the table must be aligned to 64 bytes. Otherwise the translation table must be aligned to the size of the table.
For the 64KB granule, if 52-bit PA is not supported, and the value of TCR_EL1.IPS is `0b110` or `0b111`, one of the following IMPLEMENTATION DEFINED behaviors occur:

- BADDR uses the extended format to represent a 52-bit base address.
- BADDR does not use the extended format.

If any register bit[47:1] that is defined as RES0 has the value 1 when a translation table walk is done using TTBR1_EL1, then the translation table base address might be misaligned, with effects that are CONSTRAINED UNPREDICTABLE, and must be one of the following:

- Bits A[(x-1):0] of the stage 1 translation table base address are treated as if all the bits are zero. The value read back from the corresponding register bits is either the value written to the register or zero.
- The result of the calculation of an address for a translation table walk using this register can be corrupted in those bits that are nonzero.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

CnP, bit [0]

When VTCR_EL2.MSA == '1' and FEAT_TTCNP is implemented:

Common not Private. This bit indicates whether each entry that is pointed to by TBR1_EL1 is a member of a common set that can be used by every PE in the Inner Shareable domain for which the value of TTBR1_EL1.CnP is 1.

CnP	Meaning
`0b0`	The translation table entries pointed to by TTBR1_EL1, for the current translation regime and ASID, are permitted to differ from corresponding entries for TTBR1_EL1 for other PEs in the Inner Shareable domain. This is not affected by: • The value of TTBR1_EL1.CnP on those other PEs. • The value of the current ASID. • The value of the current VMID.
`0b1`	The translation table entries pointed to by TTBR1_EL1 are the same as the translation table entries for every other PE in the Inner Shareable domain for which the value of TTBR1_EL1.CnP is 1 and all of the following apply: • The translation table entries are pointed to by TTBR1_EL1. • The translation tables relate to the same translation regime. • The ASID is the same as the current ASID. • The value of the current VMID.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This bit is permitted to be cached in a TLB.

When a TLB combines entries from stage 1 translation and stage 2 translation into a single entry, that entry can only be shared between different PEs if the value of the CnP bit is 1 for both stage 1 and stage 2.

Note

If the value of the TTBR1_EL1.CnP bit is 1 on multiple PEs in the same Inner Shareable domain and those TTBR1_EL1s do not point to the same translation table entries when the other conditions specified for the case when the value of CnP is 1 apply, then the results of translations are CONSTRAINED UNPREDICTABLE, see 'CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Accessing TTBR1_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, TTBR1_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0000	0b001

```
if !ArchHasVMSAExtension() then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if ID_AA64MMFR0_EL1.MSA == '1111' && ID_AA64MMFR0_EL1.MSA_frac == '0001' then
        UNDEFINED;
    elsif HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif VTCR_EL2.MSA == '0' then
        UNDEFINED;
    else
        X[t, 64] = TTBR1_EL1<63:0>;
elsif PSTATE.EL == EL2 then
    if ID_AA64MMFR0_EL1.MSA == '1111' && ID_AA64MMFR0_EL1.MSA_frac == '0001' then
        UNDEFINED;
    else
        X[t, 64] = TTBR1_EL1<63:0>;
```

`MSR TTBR1_EL1, <Xt>`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0010	0b0000	0b001

```
if !ArchHasVMSAExtension() then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if ID_AA64MMFR0_EL1.MSA == '1111' && ID_AA64MMFR0_EL1.MSA_frac == '0001' then
        UNDEFINED;
    elsif HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif VTCR_EL2.MSA == '0' then
        UNDEFINED;
    else
        TTBR1_EL1<63:0> = X[t, 64];
elsif PSTATE.EL == EL2 then
    if ID_AA64MMFR0_EL1.MSA == '1111' && ID_AA64MMFR0_EL1.MSA_frac == '0001' then
        UNDEFINED;
    else
        TTBR1_EL1<63:0> = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.99 VBAR_EL1, Vector Base Address Register (EL1)

The VBAR_EL1 characteristics are:

Purpose

Holds the vector base address for any exception that is taken to EL1.

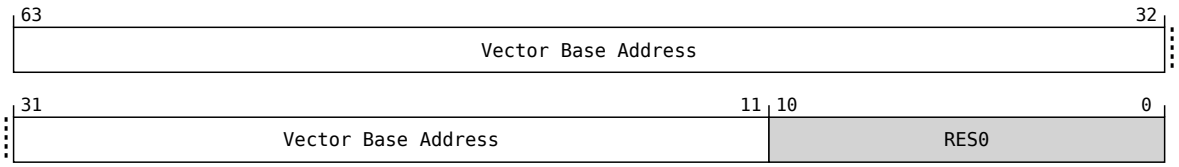
Configuration

There are no configuration notes.

Attributes

VBAR_EL1 is a 64-bit register.

Field descriptions



VBA, bits [63:11]

Vector Base Address. Base address of the exception vectors for exceptions taken to EL1.

Note

If the implementation supports FEAT_LVA, then:

- If tagged addresses are being used, bits [55:52] of VBAR_EL1 must be the same or else the use of the vector address will result in a recursive exception.
- If tagged addresses are not being used, bits [63:52] of VBAR_EL1 must be the same or else the use of the vector address will result in a recursive exception.

If the implementation does not support FEAT_LVA, then:

- If tagged addresses are being used, bits [55:48] of VBAR_EL1 must be the same or else the use of the vector address will result in a recursive exception.
- If tagged addresses are not being used, bits [63:48] of VBAR_EL1 must be the same or else the use of the vector address will result in a recursive exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [10:0]

Reserved, RES0.

Accessing VBAR_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, VBAR_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b1100	0b0000	0b000

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    X[t, 64] = VBAR_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = VBAR_EL1;
```

MSR VBAR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1100	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    VBAR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    VBAR_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.100 VBAR_EL2, Vector Base Address Register (EL2)

The VBAR_EL2 characteristics are:

Purpose

Holds the vector base address for any exception that is taken to EL2.

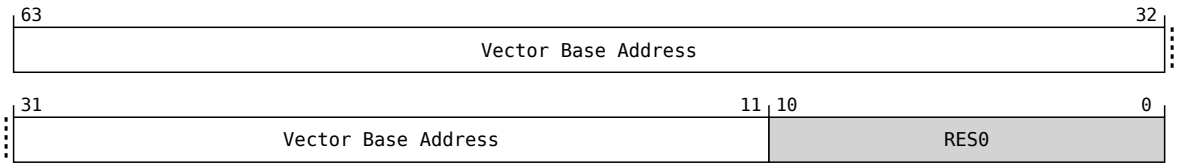
Configuration

There are no configuration notes.

Attributes

VBAR_EL2 is a 64-bit register.

Field descriptions



VBA, bits [63:11]

Vector Base Address. Base address of the exception vectors for exceptions taken to EL2.

Note

If FEAT_LVA is implemented:

```
- If tagged addresses are being used, bits [55:52] of VBAR_EL2 must be 0 or else the use of the
    ↪vector address will result in a recursive exception.
- If tagged addresses are not being used, bits [63:52] of VBAR_EL2 must be 0 or else the use of the
    ↪vector address will result in a recursive exception.
```

If FEAT_LVA is not implemented:

```
- If tagged addresses are being used, bits [55:48] of VBAR_EL2 must be 0 or else the use of the
    ↪vector address will result in a recursive exception.
- If tagged addresses are not being used, bits [63:48] of VBAR_EL2 must be 0 or else the use of
    ↪the vector address will result in a recursive exception.
```

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [10:0]

Reserved, RES0.

Accessing VBAR_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, VBAR_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1100	0b0000	0b000

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = VBAR_EL2;
```

MSR VBAR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1100	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    VBAR_EL2 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.101 VMPIDR_EL2, Virtualization Multiprocessor ID Register

The VMPIDR_EL2 characteristics are:

Purpose

Holds the value of the Virtualization Multiprocessor ID. This is the value returned by EL1 reads of MPIDR_EL1, which in a multiprocessor system, provides an additional PE identification system.

Configuration

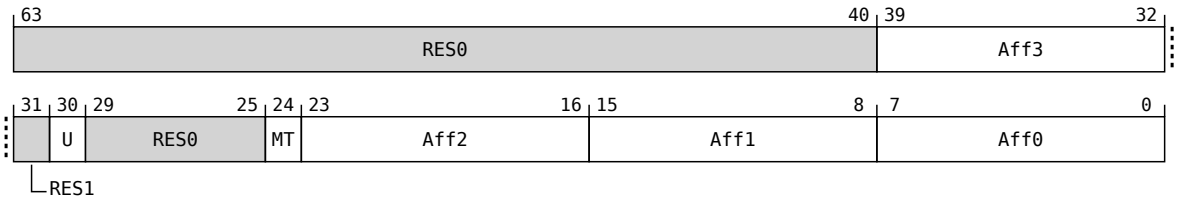
If EL2 is not implemented, reads of this register return the value of the MPIDR_EL1 and writes to the register are ignored.

This register has no effect if EL2 is not enabled in the current Security state.

Attributes

VMPIDR_EL2 is a 64-bit register.

Field descriptions



Bits [63:40]

Reserved, RES0.

Aff3, bits [39:32]

Affinity level 3. See the description of VMPIDR_EL2.Aff0 for more information.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [31]

Reserved, RES1.

U, bit [30]

Indicates a Uniprocessor system, as distinct from PE 0 in a multiprocessor system.

U	Meaning
0b0	Processor is part of a multiprocessor system.
0b1	Processor is part of a uniprocessor system.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [29:25]

Reserved, RES0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MT, bit [24]

Indicates whether the lowest level of affinity consists of logical PEs that are implemented using a multithreading type approach. See the description of VMPIDR_EL2.Aff0 for more information about affinity levels.

MT	Meaning
0b0	Performance of PEs at the lowest affinity level is largely independent.
0b1	Performance of PEs at the lowest affinity level is very interdependent.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Aff2, bits [23:16]

Affinity level 2. See the description of VMPIDR_EL2.Aff0 for more information.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Aff1, bits [15:8]

Affinity level 1. See the description of VMPIDR_EL2.Aff0 for more information.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Aff0, bits [7:0]

Affinity level 0.

The assigned value of the MPIDR_EL1.{Aff3, Aff2, Aff1, Aff0} set of fields of each PE must be unique within the system as a whole.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing VMPIDR_EL2

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, VMPIDR_EL2`

op0	op1	CRn	CRm	op2
0b11	0b100	0b0000	0b0000	0b101

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = VMPIDR_EL2;
```

`MSR VMPIDR_EL2, <Xt>`

op0	op1	CRn	CRm	op2
0b11	0b100	0b0000	0b0000	0b101

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    VMPIDR_EL2 = X[t, 64];
```

MRS <Xt>, MPIDR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0000	0b101

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    X[t, 64] = VMPIDR_EL2;
elsif PSTATE.EL == EL2 then
    X[t, 64] = MPIDR_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.102 VPIDR_EL2, Virtualization Processor ID Register

The VPIDR_EL2 characteristics are:

Purpose

Holds the value of the Virtualization Processor ID. This is the value returned by EL1 reads of MIDR_EL1.

Configuration

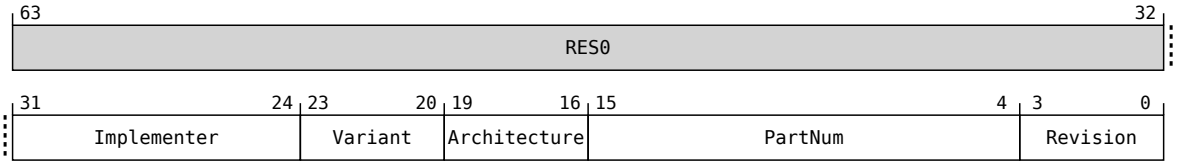
If EL2 is not implemented, reads of this register return the value of the MIDR_EL1 and writes to the register are ignored.

This register has no effect if EL2 is not enabled in the current Security state.

Attributes

VPIDR_EL2 is a 64-bit register.

Field descriptions



Bits [63:32]

Reserved, RES0.

Implementer, bits [31:24]

The Implementer code. This field must hold an implementer code that has been assigned by Arm.

Implementer	Meaning
0x00	Reserved for software use.
0x41	Arm Limited.
0x42	Broadcom Corporation.
0x43	Cavium Inc.
0x44	Digital Equipment Corporation.
0x46	Fujitsu Ltd.
0x49	Infineon Technologies AG.
0x4D	Motorola or Freescale Semiconductor Inc.
0x4E	NVIDIA Corporation.
0x50	Applied Micro Circuits Corporation.
0x51	Qualcomm Inc.
0x56	Marvell International Ltd.
0x69	Intel Corporation.
0xC0	Ampere Computing.

Arm can assign codes that are not published in this manual. All values not assigned by Arm are reserved and must not be used.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Variant, bits [23:20]

An IMPLEMENTATION DEFINED variant number. Typically, this field is used to distinguish between different product variants, or major revisions of a product.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Architecture, bits [19:16]

Architecture version.

Architecture	Meaning
0b0001	Armv4.
0b0010	Armv4T.
0b0011	Armv5 (obsolete).
0b0100	Armv5T.
0b0101	Armv5TE.
0b0110	Armv5TEJ.
0b0111	Armv6.
0b1111	Architectural features are individually identified in the ID_* registers.

All other values are reserved.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

PartNum, bits [15:4]

An IMPLEMENTATION DEFINED primary part number for the device.

On processors implemented by Arm, if the top four bits of the primary part number are 0x0 or 0x7, the variant and architecture are encoded differently.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Revision, bits [3:0]

An IMPLEMENTATION DEFINED revision number for the device.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing VPIDR_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, VPIDR_EL2
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b100	0b0000	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = VPIDR_EL2;
```

MSR VPIDR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0000	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    VPIDR_EL2 = X[t, 64];
```

MRS <Xt>, MIDR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0000	0b000

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    X[t, 64] = VPIDR_EL2;
elsif PSTATE.EL == EL2 then
    X[t, 64] = MIDR_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.103 VSTCR_EL2, Virtualization Secure Translation Control Register

The VSTCR_EL2 characteristics are:

Purpose

The control register for stage 2 of the Secure EL1&0 translation regime.

Configuration

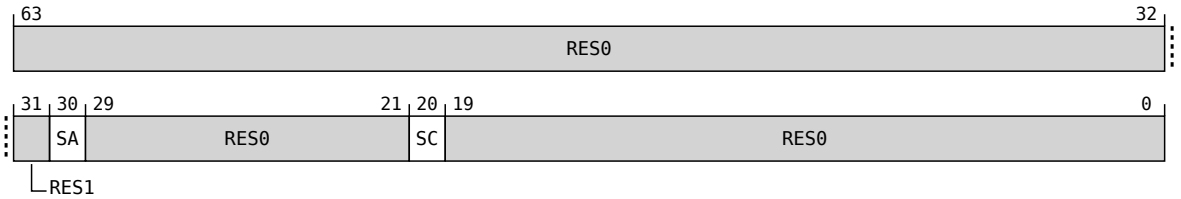
This register is present only when FEAT_SEL2 is implemented. Otherwise, direct accesses to VSTCR_EL2 are UNDEFINED.

Attributes

VSTCR_EL2 is a 64-bit register.

Field descriptions

IsProfile(PROFILE_R)



Any of the bits in VSTCR_EL2 are permitted to be cached in a TLB.

Bits [63:32]

Reserved, RES0.

Bit [31]

Reserved, RES1.

SA, bit [30]

Secure stage 2 translation output address space.

SA	Meaning
0b0	All stage 2 translations for the Secure PA space access the Secure PA space.
0b1	All stage 2 translations for the Secure PA space access the Non-secure PA space.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [29:21]

Reserved, RES0.

SC, bit [20]

NS check enable bit.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SC	Meaning
0b0	Least secure NS configuration is selected from the stage 1 and stage 2 EL1&0 translation regime for the given address.
0b1	Stage 2 NS configuration is checked against stage 1 NS configuration in EL1&0 translation regime for the given address, and generate a fault if they are different.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [19:0]

Reserved, RES0.

Accessing VSTCR_EL2

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, VSTCR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0110	0b010

```
if !IsFeatureImplemented(FEAT_SEL2) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = VSTCR_EL2;
```

MSR VSTCR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0110	0b010

```
if !IsFeatureImplemented(FEAT_SEL2) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    VSTCR_EL2 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.2.104 VTCR_EL2, Virtualization Translation Control Register

The VTCR_EL2 characteristics are:

Purpose

The control register for stage 2 of the EL1&0 translation regime.

Configuration

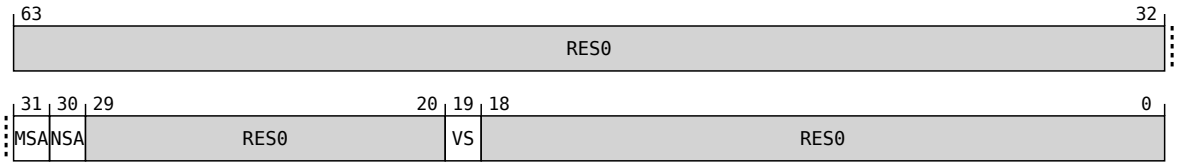
There are no configuration notes.

Attributes

VTCR_EL2 is a 64-bit register.

Field descriptions

IsProfile(PROFILE_R)



Any of the bits in VTCR_EL2 are permitted to be cached in a TLB.

Bits [63:32]

Reserved, RES0.

MSA, bit [31]

When ID_AA64MMFR0_EL1.MSA_frac == '0010':

Stage 1 EL1&0 translation regime memory system architecture.

MSA	Meaning
0b0	Stage 1 EL1&0 translation regime uses PMSAv8-64 memory architecture.
0b1	Stage 1 EL1&0 translation regime uses VMSAv8-64 memory architecture.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When ID_AA64MMFR0_EL1.MSA_frac == '0001':

Reserved, RES0.

Otherwise:

Reserved, RES1.

NSA, bit [30]

When FEAT_SEL2 is implemented:

Non-secure stage 2 translation output address space.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NSA	Meaning
0b0	All stage 2 translations for the Non-secure PA space of the Secure EL1&0 translation regime access the Secure PA space.
0b1	All stage 2 translations for the Non-secure PA space of the Secure EL1&0 translation regime access the Non-secure PA space.

This bit behaves as 1 for all purposes other than reading back the value of the bit when the value of VSTCR_EL2.SA is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [29:20]

Reserved, RES0.

VS, bit [19]

When FEAT_VMID16 is implemented:

VMID Size.

VS	Meaning
0b0	8-bit VMID. The upper 8 bits of VSCTLR_EL2 are ignored by the hardware, and treated as if they are all zeros, for every purpose except when reading back the register.
0b1	16-bit VMID. The upper 8 bits of VSCTLR_EL2 are used for allocation and matching in the TLB.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [18:0]

Reserved, RES0.

Accessing VTCR_EL2

Unless stated otherwise, any of the bits in VTCR_EL2 are permitted to be cached in a TLB.

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, VTCR_EL2`

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0001	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = VTCR_EL2;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR VTCR_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0010	0b0001	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    VTCR_EL2 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.3 Debug registers

This section lists the Debug System registers in AArch64 state, in alphabetic order:

- The principal encoding space for debug registers is `op0` == `0b10`, `op1` == {0, 3, 4}. Instructions for accessing debug System registers summarizes the registers in this encoding space and lists them in order of their encodings.
- In addition, the following registers in the `op0` == `0b11` encoding space are classified as Debug registers:
  - DLR_EL0.
  - DSPSR_EL0.
  - MDCR_EL2.
  - MDCR_EL3.
  - SDER32_EL3.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.3.1 DBGAUTHSTATUS_EL1, Debug Authentication Status Register

The DBGAUTHSTATUS_EL1 characteristics are:

Purpose

Provides information about the state of the IMPLEMENTATION DEFINED authentication interface for debug.

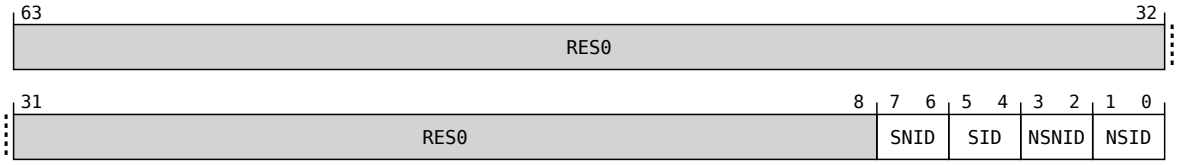
Configuration

AArch64 System register DBGAUTHSTATUS_EL1 bits [31:0] are architecturally mapped to External register DBGAUTHSTATUS_EL1[31:0].

Attributes

DBGAUTHSTATUS_EL1 is a 64-bit register.

Field descriptions



Bits [63:8]

Reserved, RES0.

SNID, bits [7:6]

When FEAT_Debugv8p4 is implemented:

Secure non-invasive debug.

This field has the same value as DBGAUTHSTATUS_EL1.SID.

Otherwise:

Secure non-invasive debug.

SNID	Meaning
0b00	Secure state is not implemented.
0b10	Implemented and disabled. ExternalSecureNoninvasiveDebugEnabled() == FALSE.
0b11	Implemented and enabled. ExternalSecureNoninvasiveDebugEnabled() == TRUE.

All other values are reserved.

SID, bits [5:4]

Secure invasive debug.

SID	Meaning
0b00	Secure state is not implemented.
0b10	Implemented and disabled. ExternalSecureInvasiveDebugEnabled() == FALSE.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b11	Implemented and enabled. ExternalSecureInvasiveDebugEnabled() == TRUE.

All other values are reserved.

NSNID, bits [3:2]

When FEAT_Debugv8p4 is implemented:

Non-secure non-invasive debug.

NSNID	Meaning
0b00	Non-secure state is not implemented.
0b11	Implemented and enabled. EL3 is implemented or the Effective value of SCR_EL3.NS is 1.

All other values are reserved.

Otherwise:

Non-secure non-invasive debug.

NSNID	Meaning
0b00	Non-secure state is not implemented.
0b10	Implemented and disabled. ExternalNoninvasiveDebugEnabled() == FALSE.
0b11	Implemented and enabled. ExternalNoninvasiveDebugEnabled() == TRUE.

All other values are reserved.

NSID, bits [1:0]

Non-secure invasive debug.

NSID	Meaning
0b00	Non-secure state is not implemented.
0b10	Implemented and disabled. ExternalInvasiveDebugEnabled() == FALSE.
0b11	Implemented and enabled. ExternalInvasiveDebugEnabled() == TRUE.

All other values are reserved.

Accessing DBGAUTHSTATUS_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, DBGAUTHSTATUS_EL1`

op0	op1	CRn	CRm	op2
0b10	0b000	0b0111	0b1110	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        X[t, 64] = DBGAUTHSTATUS_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = DBGAUTHSTATUS_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.3.2 DBGBCR<n>_EL1, Debug Breakpoint Control Registers, n = 0 - 63

The DBGBCR<n>_EL1 characteristics are:

Purpose

Holds control information for a breakpoint. Forms breakpoint n together with value register DBGBVR<n>_EL1.

Configuration

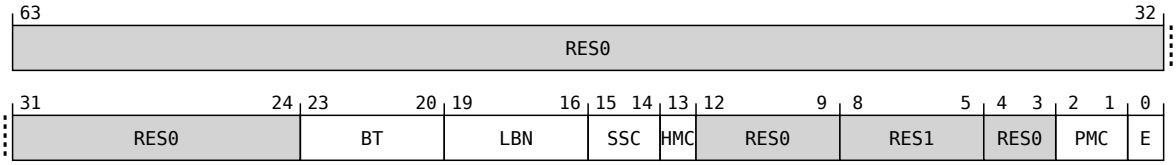
If breakpoint n is not implemented, accesses to this register are UNDEFINED.

AArch64 System register DBGBCR<n>_EL1 bits [31:0] are architecturally mapped to External register DBGBCR<n>_EL1[31:0].

Attributes

DBGBCR<n>_EL1 is a 64-bit register.

Field descriptions



Bits [63:24]

Reserved, RES0.

BT, bits [23:20]

Breakpoint Type.

Specifies breakpoint type.

BT	Meaning	Applies when
0b0000	Unlinked instruction address match. DBGBVR<n>_EL1 is the address of an instruction.	
0b0001	Linked instruction address match. As 0b0000, but linked to a breakpoint that has linking enabled.	
0b0010	Unlinked Context ID match. If the Effective value of HCR_EL2.E2H is 1, and either the PE is executing at EL0 with HCR_EL2.TGE set to 1 or the PE is executing at EL2, then DBGBVR<n>_EL1.ContextID is compared against CONTEXTIDR_EL2. Otherwise, DBGBVR<n>_EL1.ContextID is compared against CONTEXTIDR_EL1.	breakpoint n is context-aware
0b0011	As 0b0010, with linking enabled.	breakpoint n is context-aware
0b0110	Unlinked CONTEXTIDR_EL1 match. DBGBVR<n>_EL1.ContextID is a Context ID compared against CONTEXTIDR_EL1.	FEAT_Debugv8p1 is implemented and breakpoint n is context-aware
0b0111	As 0b0110, with linking enabled.	FEAT_Debugv8p1 is implemented and breakpoint n is context-aware

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

BT	Meaning	Applies when
0b1000	Unlinked VMID match. DBGBVR<n>_EL1.VMID is a VMID compared against VTTBR_EL2.VMID.	breakpoint n is context-aware
0b1001	As 0b1000, with linking enabled.	breakpoint n is context-aware
0b1010	Unlinked VMID and Context ID match. DBGBVR<n>_EL1.ContextID is a Context ID compared against CONTEXTIDR_EL1, and DBGBVR<n>_EL1.VMID is a VMID compared against VSCTLR_EL2.VMID.	breakpoint n is context-aware
0b1011	As 0b1010, with linking enabled.	breakpoint n is context-aware
0b1100	Unlinked CONTEXTIDR_EL2 match. DBGBVR<n>_EL1.ContextID2 is a Context ID compared against CONTEXTIDR_EL2.	FEAT_Debugv8p1 is implemented and breakpoint n is context-aware
0b1101	As 0b1100, with linking enabled.	FEAT_Debugv8p1 is implemented and breakpoint n is context-aware
0b1110	Unlinked Full Context ID match. DBGBVR<n>_EL1.ContextID is compared against CONTEXTIDR_EL1, and DBGBVR<n>_EL1.ContextID2 is compared against CONTEXTIDR_EL2.	FEAT_Debugv8p1 is implemented and breakpoint n is context-aware
0b1111	As 0b1110, with linking enabled.	FEAT_Debugv8p1 is implemented and breakpoint n is context-aware

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

LBN, bits [19:16]

Linked Breakpoint Number.

For Linked address matching breakpoints, specifies the index of the breakpoint linked to.

For all other breakpoint types, this field is ignored and reads of the register return an UNKNOWN value.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

SSC, bits [15:14]

Security state control. Determines the Security states under which a Breakpoint debug event for breakpoint n is generated.

The fields that indicate when the breakpoint can be generated are: HMC, PMC, and SSC. These fields must be considered in combination, and the values that are permitted for these fields are constrained.

For more information on the operation of these fields, see 'Execution conditions for which a breakpoint generates Breakpoint exceptions'.

For more information on the effect of programming the fields to a reserved set of values, see 'Reserved DBGBCR<n>_EL1.{SSC, HMC, PMC} values'.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

HMC, bit [13]

Higher mode control. Determines the debug perspective for deciding when a Breakpoint debug event for breakpoint n is generated.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The fields that indicate when the breakpoint can be generated are: HMC, PMC, and SSC. These fields must be considered in combination, and the values that are permitted for these fields are constrained.

For more information on the operation of these fields, see 'Execution conditions for which a breakpoint generates Breakpoint exceptions'.

For more information, see DBGBCR<n>_EL1.SSC.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [12:9]

Reserved, RES0.

Bits [8:5]

Reserved, RES1.

Bits [4:3]

Reserved, RES0.

PMC, bits [2:1]

Privilege mode control. Determines the Exception level or levels at which a Breakpoint debug event for breakpoint n is generated.

The fields that indicate when the breakpoint can be generated are: HMC, PMC, and SSC. These fields must be considered in combination, and the values that are permitted for these fields are constrained.

For more information on the operation of these fields, see 'Execution conditions for which a breakpoint generates Breakpoint exceptions'.

For more information, see DBGBCR<n>_EL1.SSC.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

E, bit [0]

Enable breakpoint n.

E	Meaning
0b0	Breakpoint n disabled.
0b1	Breakpoint n enabled.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing DBGBCR<n>_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, DBGBCR<m>_EL1 ; Where m = 0-15
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	m[3:0]	0b101

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
integer m = UInt(CRm<3:0>);

if m >= NUM_BREAKPOINTS then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif OSLSR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X[t, 64] = DBGBCR_EL1[m];
elsif PSTATE.EL == EL2 then
    if OSLSR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X[t, 64] = DBGBCR_EL1[m];
```

MSR DBGBCR<m>_EL1, <Xt> ; Where m = 0-15

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	m[3:0]	0b101

```
integer m = UInt(CRm<3:0>);

if m >= NUM_BREAKPOINTS then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif OSLSR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGBCR_EL1[m] = X[t, 64];
elsif PSTATE.EL == EL2 then
    if OSLSR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGBCR_EL1[m] = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.3.3 DBGBVR<n>_EL1, Debug Breakpoint Value Registers, n = 0 - 63

The DBGBVR<n>_EL1 characteristics are:

Purpose

Holds a virtual address, or a VMID and/or a context ID, for use in breakpoint matching. Forms breakpoint n together with control register DBGBCR<n>_EL1.

Configuration

How this register is interpreted depends on the value of DBGBCR<n>_EL1.BT.

- When DBGBCR<n>_EL1.BT is `0b000x`, this register holds a virtual address.
- When DBGBCR<n>_EL1.BT is `0b001x`, `0b011x`, or `0b110x`, this register holds a Context ID.
- When DBGBCR<n>_EL1.BT is `0b100x`, this register holds a VMID.
- When DBGBCR<n>_EL1.BT is `0b101x`, this register holds a VMID and a Context ID.
- When DBGBCR<n>_EL1.BT is `0b111x`, this register holds two Context ID values.

For other values of DBGBCR<n>_EL1.BT, this register is RES0.

If breakpoint n is not implemented then accesses to this register are UNDEFINED.

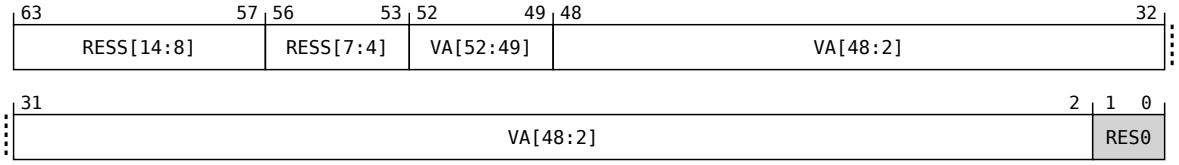
AArch64 System register DBGBVR<n>_EL1 bits [63:0] are architecturally mapped to External register DBGBVR<n>_EL1[63:0].

Attributes

DBGBVR<n>_EL1 is a 64-bit register.

Field descriptions

When DBGBCR<n>_EL1.BT IN {'000x'}:



RESS[14:8], bits [63:57]

Reserved, Sign extended. Software must set all bits in this field to the same value as the most significant bit of the VA field. If all bits in this field are not the same value as the most significant bit of the VA field, then all of the following apply:

- It is CONSTRAINED UNPREDICTABLE whether the PE ignores this field when comparing an address.
- If the breakpoint is not context-aware, it is IMPLEMENTATION DEFINED whether the value read back in each bit of this field is a copy of the most significant bit of the VA field or the value written.

RESS[7:4], bits [56:53]

Extension to RESS[14:8]. For more information, see RESS[14:8].

Bits [52:49]

When FEAT_LVA is implemented:

VA[52:49], bits [52:49]

Extension to VA[48:2]. For more information, see VA[48:2].

The reset behavior of this field is:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RESS[3:0], bits [52:49]

Extension to RESS[14:8]. For more information, see RESS[14:8].

VA[48:2], bits [48:2]

Bits[48:2] of the address value for comparison.

When FEAT_LVA is implemented, VA[52:49] forms the upper part of the address value. If FEAT_LVA is not implemented, bits [52:49] are part of the RESS field.

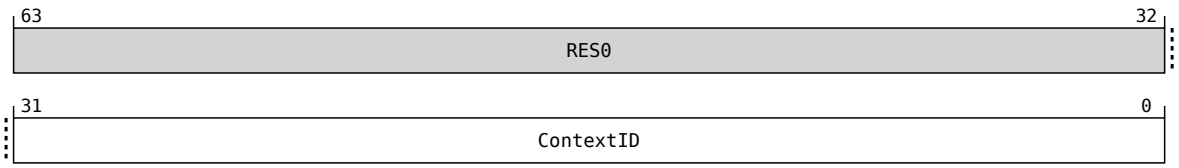
The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [1:0]

Reserved, RES0.

When DBGBCR<n>_EL1.BT IN {'001x'}:



Bits [63:32]

Reserved, RES0.

ContextID, bits [31:0]

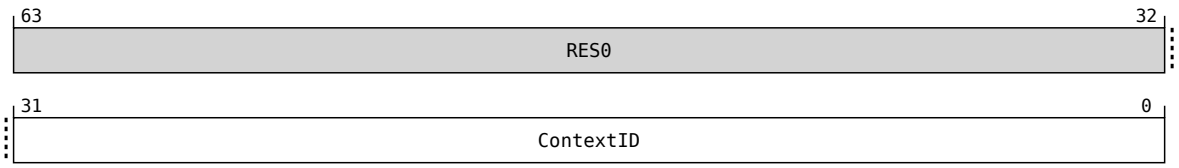
Context ID value for comparison.

The value is compared against CONTEXTIDR_EL1.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

When DBGBCR<n>_EL1.BT IN {'011x'} and FEAT_Debugv8p1 is implemented:



Bits [63:32]

Reserved, RES0.

ContextID, bits [31:0]

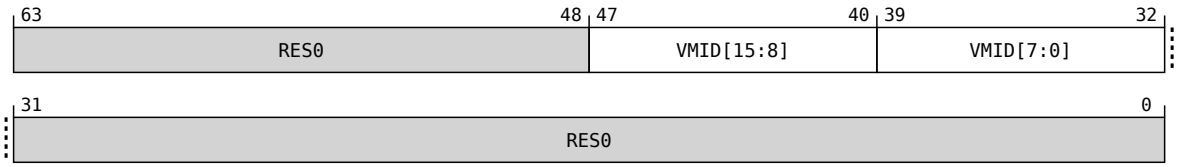
Context ID value for comparison against CONTEXTIDR_EL1.

The reset behavior of this field is:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

When DBGBCR<n>_EL1.BT IN {'100x'}:



Bits [63:48]

Reserved, RES0.

VMID[15:8], bits [47:40]

When FEAT_VMID16 is implemented and VTCR_EL2.VS == '1':

Extension to VMID[7:0]. For more information, see DBGBVR<n>_EL1.VMID[7:0].

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

VMID[7:0], bits [39:32]

VMID value for comparison.

The VMID is 8 bits when any of the following are true:

- VTCR_EL2.VS is 0.
- FEAT_VMID16 is not implemented.

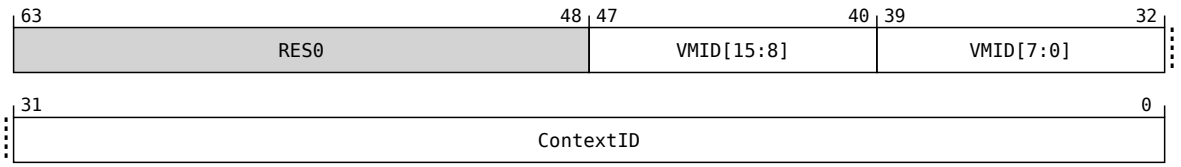
The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [31:0]

Reserved, RES0.

When DBGBCR<n>_EL1.BT IN {'101x'}:



Bits [63:48]

Reserved, RES0.

VMID[15:8], bits [47:40]

When FEAT_VMID16 is implemented and VTCR_EL2.VS == '1':

Extension to VMID[7:0]. For more information, see DBGBVR<n>_EL1.VMID[7:0].

The reset behavior of this field is:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

VMID[7:0], bits [39:32]

VMID value for comparison.

The VMID is 8 bits when any of the following are true:

- VTCR_EL2.VS is 0.
- FEAT_VMID16 is not implemented.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

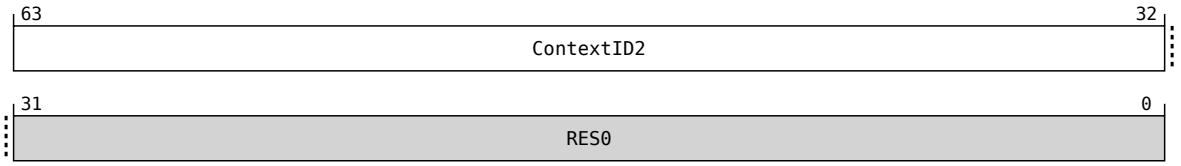
ContextID, bits [31:0]

Context ID value for comparison against CONTEXTIDR_EL1.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

When DBGBCR<n>_EL1.BT IN {'110x'} and FEAT_Debugv8p1 is implemented:



ContextID2, bits [63:32]

Context ID value for comparison against CONTEXTIDR_EL2.

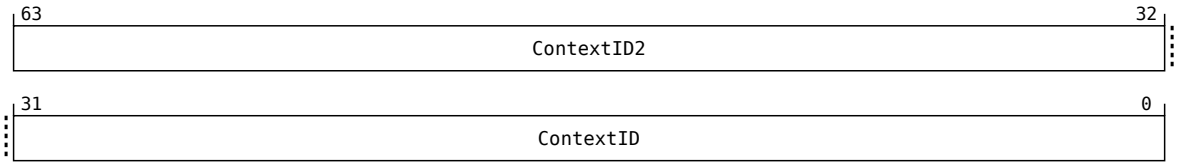
The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [31:0]

Reserved, RES0.

When DBGBCR<n>_EL1.BT IN {'111x'} and FEAT_Debugv8p1 is implemented:



ContextID2, bits [63:32]

Context ID value for comparison against CONTEXTIDR_EL2.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ContextID, bits [31:0]

Context ID value for comparison against CONTEXTIDR_EL1.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing DBGBVR<n>_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, DBGBVR<m>_EL1 ; Where m = 0-15
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	m[3:0]	0b100

```
integer m = UInt(CRm<3:0>);

if m >= NUM_BREAKPOINTS then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif OSLSR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X[t, 64] = DBGBVR_EL1[m];
elsif PSTATE.EL == EL2 then
    if OSLSR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X[t, 64] = DBGBVR_EL1[m];
```

```
MSR DBGBVR<m>_EL1, <Xt> ; Where m = 0-15
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	m[3:0]	0b100

```
integer m = UInt(CRm<3:0>);

if m >= NUM_BREAKPOINTS then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif OSLSR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGBVR_EL1[m] = X[t, 64];
elsif PSTATE.EL == EL2 then
    if OSLSR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGBVR_EL1[m] = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.3.4 DBGCLAIMCLR_EL1, Debug CLAIM Tag Clear Register

The DBGCLAIMCLR_EL1 characteristics are:

Purpose

Used by software to read the values of the CLAIM tag bits, and to clear CLAIM tag bits to 0.

The architecture does not define any functionality for the CLAIM tag bits.

Note

CLAIM tags are typically used for communication between the debugger and target software.

Used in conjunction with the DBGCLAIMSET_EL1 register.

Configuration

An implementation must include eight CLAIM tag bits.

AArch64 System register DBGCLAIMCLR_EL1 bits [63:0] are architecturally mapped to AArch64 System register DBGCLAIMSET_EL1[63:0].

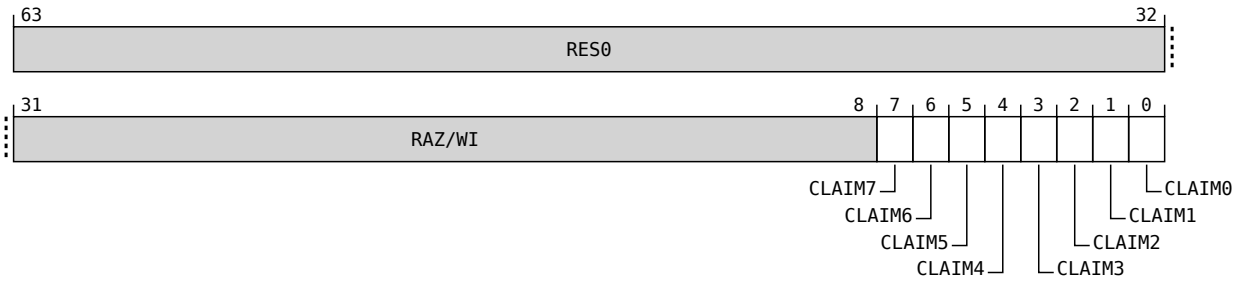
AArch64 System register DBGCLAIMCLR_EL1 bits [31:0] are architecturally mapped to External register DBGCLAIMCLR_EL1[31:0].

AArch64 System register DBGCLAIMCLR_EL1 bits [31:0] are architecturally mapped to External register DBGCLAIMSET_EL1[31:0].

Attributes

DBGCLAIMCLR_EL1 is a 64-bit register.

Field descriptions



Bits [63:32]

Reserved, RES0.

Bits [31:8]

Reserved, RAZ/WI.

CLAIM<m> , bits [m], for m = 7 to 0

Claim Tag Clear. Indicates the current status of Claim Tag bit <m>, and is used to clear Claim Tag bit <m> to 0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CLAIM<m>	Meaning
0b0	On a read: Claim Tag bit <m> is not set. On a write: Ignored.
0b1	On a read: Claim Tag bit <m> is set. On a write: Clear Claim tag bit <m> to 0.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Access to this field is W1C.

Accessing DBGCLAIMCLR_EL1

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, DBGCLAIMCLR_EL1

op0	op1	CRn	CRm	op2
0b10	0b000	0b0111	0b1001	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = DBGCLAIMCLR_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = DBGCLAIMCLR_EL1;
```

MSR DBGCLAIMCLR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b10	0b000	0b0111	0b1001	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        DBGCLAIMCLR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    DBGCLAIMCLR_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.3.5 DBGCLAIMSET_EL1, Debug CLAIM Tag Set Register

The DBGCLAIMSET_EL1 characteristics are:

Purpose

Used by software to set the CLAIM tag bits to 1.

The architecture does not define any functionality for the CLAIM tag bits.

Note

CLAIM tags are typically used for communication between the debugger and target software.

Used in conjunction with the DBGCLAIMCLR_EL1 register.

Configuration

An implementation must include eight CLAIM tag bits.

AArch64 System register DBGCLAIMSET_EL1 bits [63:0] are architecturally mapped to AArch64 System register DBGCLAIMCLR_EL1[63:0].

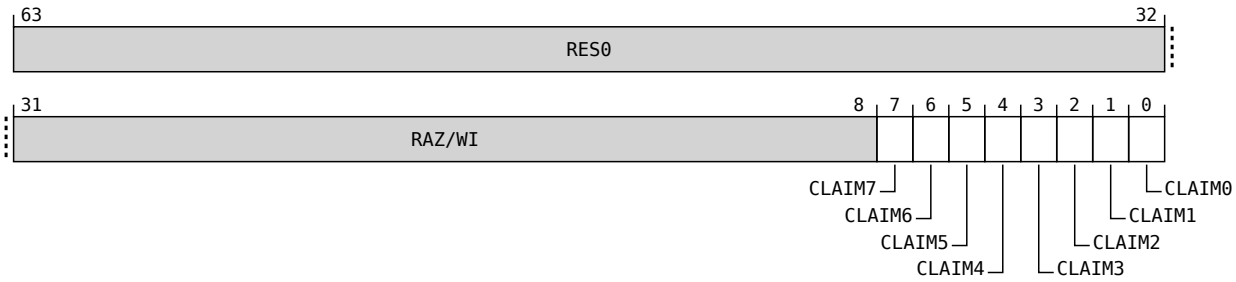
AArch64 System register DBGCLAIMSET_EL1 bits [31:0] are architecturally mapped to External register DBGCLAIMSET_EL1[31:0].

AArch64 System register DBGCLAIMSET_EL1 bits [31:0] are architecturally mapped to External register DBGCLAIMCLR_EL1[31:0].

Attributes

DBGCLAIMSET_EL1 is a 64-bit register.

Field descriptions



Bits [63:32]

Reserved, RES0.

Bits [31:8]

Reserved, RAZ/WI.

CLAIM<m> , bits [m], for m = 7 to 0

Claim Tag Set. Used to set Claim Tag bit <m> to 1.

CLAIM<m>	Meaning
0b0	On a write: Ignored.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CLAIM<m>	Meaning
0b1	On a write: Set Claim Tag bit <m> to 1.

Access to this field is RAO/W1S.

Accessing DBGCLAIMSET_EL1

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, DBGCLAIMSET_EL1

op0	op1	CRn	CRm	op2
0b10	0b000	0b0111	0b1000	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = DBGCLAIMSET_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = DBGCLAIMSET_EL1;
```

MSR DBGCLAIMSET_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b10	0b000	0b0111	0b1000	0b110

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        DBGCLAIMSET_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    DBGCLAIMSET_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.3.6 DBGDTR_EL0, Debug Data Transfer Register, half-duplex

The DBGDTR_EL0 characteristics are:

Purpose

Transfers 64 bits of data between the PE and an external debugger. Can transfer both ways using only a single register.

Configuration

When written, AArch64 System register DBGDTR_EL0 bits [63:32] are architecturally mapped to External register DBGDTRRX_EL0[31:0].

When written, AArch64 System register DBGDTR_EL0 bits [63:32] are architecturally mapped to AArch64 System register DBGDTRRX_EL0[31:0].

When written, AArch64 System register DBGDTR_EL0 bits [31:0] are architecturally mapped to External register DBGDTRTX_EL0[31:0].

When written, AArch64 System register DBGDTR_EL0 bits [31:0] are architecturally mapped to AArch64 System register DBGDTRTX_EL0[31:0].

When read, AArch64 System register DBGDTR_EL0 bits [63:32] are architecturally mapped to External register DBGDTRTX_EL0[31:0].

When read, AArch64 System register DBGDTR_EL0 bits [63:32] are architecturally mapped to AArch64 System register DBGDTRTX_EL0[31:0].

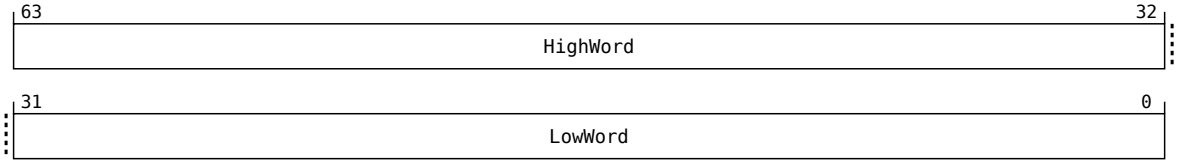
When read, AArch64 System register DBGDTR_EL0 bits [31:0] are architecturally mapped to External register DBGDTRRX_EL0[31:0].

When read, AArch64 System register DBGDTR_EL0 bits [31:0] are architecturally mapped to AArch64 System register DBGDTRRX_EL0[31:0].

Attributes

DBGDTR_EL0 is a 64-bit register.

Field descriptions



HighWord, bits [63:32]

Writes to this register set DTRRX to the value in this field and do not change RXfull.

Reads of this register:

- If RXfull is 1, return the last value written to DTRTX.
- If RXfull is 0, return an UNKNOWN value.

After the read, RXfull is cleared to 0.

LowWord, bits [31:0]

Writes to this register set DTRTX to the value in this field and set TXfull to 1.

Reads of this register:

- If RXfull is 1, return the last value written to DTRRX.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If RXfull is 0, return an UNKNOWN value.

After the read, RXfull is cleared to 0.

Accessing DBGDTR_EL0

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, DBGDTR_EL0

op0	op1	CRn	CRm	op2
0b10	0b011	0b0000	0b0100	0b000

```
if Halted() then
    X[t, 64] = Read_DBGDTR_EL0(64);
elsif PSTATE.EL == EL0 then
    if MDSCR_EL1.TDCC == '1' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif HCR_EL2.TGE == '1' || MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = Read_DBGDTR_EL0(64);
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = Read_DBGDTR_EL0(64);
elsif PSTATE.EL == EL2 then
    X[t, 64] = Read_DBGDTR_EL0(64);
```

MSR DBGDTR_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b10	0b011	0b0000	0b0100	0b000

```
if Halted() then
    Write_DBGDTR_EL0(X[t, 64]);
elsif PSTATE.EL == EL0 then
    if MDSCR_EL1.TDCC == '1' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif HCR_EL2.TGE == '1' || MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        Write_DBGDTR_EL0(X[t, 64]);
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        Write_DBGDTR_EL0(X[t, 64]);
elsif PSTATE.EL == EL2 then
    Write_DBGDTR_EL0(X[t, 64]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.3.7 DBGDTRRX_EL0, Debug Data Transfer Register, Receive

The DBGDTRRX_EL0 characteristics are:

Purpose

Transfers data from an external debugger to the PE. For example, it is used by a debugger transferring commands and data to a debug target. See DBGDTR_EL0 for additional architectural mappings. It is a component of the Debug Communications Channel.

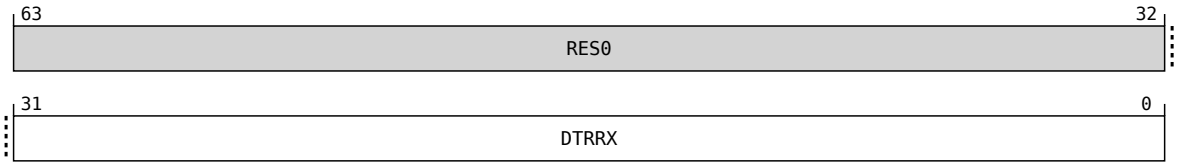
Configuration

AArch64 System register DBGDTRRX_EL0 bits [31:0] are architecturally mapped to External register DBGDTRRX_EL0[31:0].

Attributes

DBGDTRRX_EL0 is a 64-bit register.

Field descriptions



Bits [63:32]

Reserved, RES0.

DTRRX, bits [31:0]

DTRRX. Reads of this register:

- If RXfull is 1, return the last value written to DTRRX.
- If RXfull is 0, return an UNKNOWN value.

After the read, RXfull is cleared to 0.

For the full behavior of the Debug Communications Channel, see 'The Debug Communication Channel and Instruction Transfer Register'.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing DBGDTRRX_EL0

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, DBGDTRRX_EL0
```

op0	op1	CRn	CRm	op2
0b10	0b011	0b0000	0b0101	0b000

```
if Halted() then
    X[t, 32] = Read_DBGDTR_EL0(32);
elsif PSTATE.EL == EL0 then
    if MDSCR_EL1.TDCC == '1' then
        if HCR_EL2.TGE == '1' then
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif HCR_EL2.TGE == '1' || MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 32] = Read_DBGDTR_EL0(32);
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 32] = Read_DBGDTR_EL0(32);
elsif PSTATE.EL == EL2 then
    X[t, 32] = Read_DBGDTR_EL0(32);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.3.8 DBGDTRTX_EL0, Debug Data Transfer Register, Transmit

The DBGDTRTX_EL0 characteristics are:

Purpose

Transfers data from the PE to an external debugger. For example, it is used by a debug target to transfer data to the debugger. See DBGDTR_EL0 for additional architectural mappings. It is a component of the Debug Communication Channel.

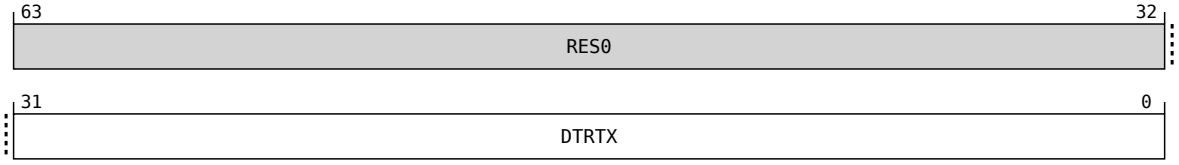
Configuration

AArch64 System register DBGDTRTX_EL0 bits [31:0] are architecturally mapped to External register DBGDTRTX_EL0[31:0].

Attributes

DBGDTRTX_EL0 is a 64-bit register.

Field descriptions



Bits [63:32]

Reserved, RES0.

DTRTX, bits [31:0]

DTRTX. Writes to this register:

- If TXfull is 1, DTRTX is set to an UNKNOWN value.
- If TXfull is 0, update the value in DTRTX.

After the write, TXfull is set to 1.

For the full behavior of the Debug Communications Channel, see 'The Debug Communication Channel and Instruction Transfer Register'.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing DBGDTRTX_EL0

Accesses to this register use the following encodings in the System register encoding space:

```
MSR DBGDTRTX_EL0, <Xt>
```

op0	op1	CRn	CRm	op2
0b10	0b011	0b0000	0b0101	0b000

```
if Halted() then
    Write_DBGDTR_EL0(X[t, 32]);
elsif PSTATE.EL == EL0 then
    if MDSCR_EL1.TDCC == '1' then
        if HCR_EL2.TGE == '1' then
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif HCR_EL2.TGE == '1' || MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        Write_DBGDTR_EL0(X[t, 32]);
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        Write_DBGDTR_EL0(X[t, 32]);
elsif PSTATE.EL == EL2 then
    Write_DBGDTR_EL0(X[t, 32]);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.3.9 DBGPRCR_EL1, Debug Power Control Register

The DBGPRCR_EL1 characteristics are:

Purpose

Controls behavior of the PE on powerdown request.

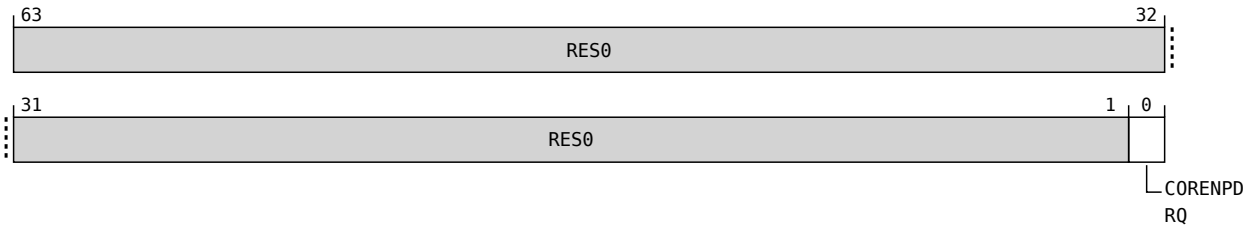
Configuration

AArch64 System register DBGPRCR_EL1 bit [0] is architecturally mapped to External register EDPRCR[0].

Attributes

DBGPRCR_EL1 is a 64-bit register.

Field descriptions



Bits [63:1]

Reserved, RES0.

CORENPDRQ, bit [0]

When FEAT_DoPD is implemented:

Core no powerdown request. Requests emulation of powerdown.

This request is typically passed to an external power controller. This means that whether a request causes power up is dependent on the IMPLEMENTATION DEFINED nature of the system. The power controller must not allow the Core power domain to switch off while this bit is 1.

CORENPDRQ	Meaning
0b0	If the system responds to a powerdown request, it powers down Core power domain.
0b1	If the system responds to a powerdown request, it does not powerdown the Core power domain, but instead emulates a powerdown of that domain.

In an implementation that includes the recommended external debug interface, this bit drives the **DBGNOPWRDWN** signal.

It is IMPLEMENTATION DEFINED whether this bit is reset to its Cold reset value on exit from an IMPLEMENTATION DEFINED software-visible retention state. For more information about retention states see 'Core power domain power states'.

Note

Writes to this bit are not prohibited by the IMPLEMENTATION DEFINED authentication interface. This means that a debugger can request emulation of powerdown regardless of whether invasive debug is permitted.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

On a Cold reset, if the powerup request is implemented and the powerup request has been asserted, this field is set to an IMPLEMENTATION DEFINED choice of 0 or 1. If the powerup request is not asserted, this field is set to 0.

Otherwise:

Core no powerdown request. Requests emulation of powerdown.

This request is typically passed to an external power controller. This means that whether a request causes power up is dependent on the IMPLEMENTATION DEFINED nature of the system. The power controller must not allow the Core power domain to switch off while this bit is 1.

CORENPDRQ	Meaning
0b0	If the system responds to a powerdown request, it powers down Core power domain.
0b1	If the system responds to a powerdown request, it does not powerdown the Core power domain, but instead emulates a powerdown of that domain.

In an implementation that includes the recommended external debug interface, this bit drives the **DBGNOPWRDWN** signal.

It is IMPLEMENTATION DEFINED whether this bit is reset to the value of EDPRCR.COREPURQ on exit from an IMPLEMENTATION DEFINED software-visible retention state. For more information about retention states see 'Core power domain power states'.

Note

Writes to this bit are not prohibited by the IMPLEMENTATION DEFINED authentication interface. This means that a debugger can request emulation of powerdown regardless of whether invasive debug is permitted.

The reset behavior of this field is:

- On a Cold reset, this field resets to the value in EDPRCR.COREPURQ.

Accessing DBGPRCR_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, DBGPRCR_EL1`

op0	op1	CRn	CRm	op2
0b10	0b000	0b0001	0b0100	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.<TDE,TDOSA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = DBGPRCR_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = DBGPRCR_EL1;
```

`MSR DBGPRCR_EL1, <Xt>`

op0	op1	CRn	CRm	op2
0b10	0b000	0b0001	0b0100	0b100

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.<TDE,TDOSA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        DBGPRCR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    DBGPRCR_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.3.10 DBGWCR<n>_EL1, Debug Watchpoint Control Registers, n = 0 - 63

The DBGWCR<n>_EL1 characteristics are:

Purpose

Holds control information for a watchpoint. Forms watchpoint n together with value register DBGWVR<n>_EL1.

Configuration

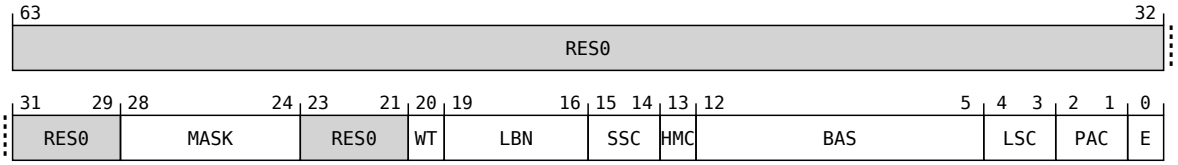
If watchpoint n is not implemented then accesses to this register are UNDEFINED.

AArch64 System register DBGWCR<n>_EL1 bits [31:0] are architecturally mapped to External register DBGWCR<n>_EL1[31:0].

Attributes

DBGWCR<n>_EL1 is a 64-bit register.

Field descriptions



Bits [63:29]

Reserved, RES0.

MASK, bits [28:24]

Address Mask. Only address ranges up to 2GB can be watched using a single mask.

MASK	Meaning
0b00000	No mask.
0b00011..0b11111	Number of address bits masked.

All other values are reserved.

Indicates the number of masked address bits, from 0b00011 masking 3 address bits (0x00000007 mask for address) to 0b11111 masking 31 address bits (0x7FFFFFFF mask for address).

If programmed with a reserved value, the watchpoint behaves as if one of the following:

- DBGWCR<n>_EL1.MASK has been programmed with a defined value, which might be 0b00000 (no mask), other than for a direct read of DBGWCR<n>_EL1.
- The watchpoint is disabled.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [23:21]

Reserved, RES0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

WT, bit [20]

Watchpoint type. Possible values are:

WT	Meaning
0b0	Unlinked watchpoint.
0b1	Linked watchpoint.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

LBN, bits [19:16]

Linked Breakpoint Number.

For Linked data address watchpoints, specifies the index of the breakpoint linked to.

For all other watchpoint types, this field is ignored and reads of the register return an UNKNOWN value.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

SSC, bits [15:14]

Security state control. Determines the Security states under which a Watchpoint debug event for watchpoint n is generated.

The fields that indicate when the watchpoint can be generated are: HMC, PAC, and SSC. These fields must be considered in combination, and the values that are permitted for these fields are constrained.

For more information on the operation of these fields, see 'Execution conditions for which a watchpoint generates Watchpoint exceptions'.

For more information on the effect of programming the fields to a reserved value, see 'Reserved DBGWCR<n>_EL1.{SSC, HMC, PAC} values'.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

HMC, bit [13]

Higher mode control. Determines the debug perspective for deciding when a Watchpoint debug event for watchpoint n is generated.

The fields that indicate when the watchpoint can be generated are: HMC, PAC, and SSC. These fields must be considered in combination, and the values that are permitted for these fields are constrained.

For more information on the operation of these fields, see 'Execution conditions for which a watchpoint generates Watchpoint exceptions'.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

BAS, bits [12:5]

Byte address select. Each bit of this field selects whether a byte from within the word or double-word addressed by DBGWVR<n>_EL1 is being watched.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

BAS	Description
xxxxxxx1	Match byte at DBGWVR<n>_EL1
xxxxxx1x	Match byte at DBGWVR<n>_EL1 + 1
xxxxx1xx	Match byte at DBGWVR<n>_EL1 + 2
xxxx1xxx	Match byte at DBGWVR<n>_EL1 + 3

In cases where DBGWVR<n>_EL1 addresses a double-word:

BAS	Description, if DBGWVR<n>_EL1[2] == 0
xxx1xxxx	Match byte at DBGWVR<n>_EL1 + 4
xx1xxxxx	Match byte at DBGWVR<n>_EL1 + 5
x1xxxxxx	Match byte at DBGWVR<n>_EL1 + 6
1xxxxxxx	Match byte at DBGWVR<n>_EL1 + 7

If DBGWVR<n>_EL1[2] == 1, only BAS[3:0] are used and BAS[7:4] are ignored. Arm deprecates setting DBGWVR<n>_EL1[2] == 1.

The valid values for BAS are nonzero binary numbers all of whose set bits are contiguous. All other values are reserved and must not be used by software. See 'Reserved DBGWCR<n>_EL1.BAS values'.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

LSC, bits [4:3]

Load/store control. This field enables watchpoint matching on the type of access being made. Possible values of this field are:

LSC	Meaning
0b01	Match instructions that load from a watchpointed address.
0b10	Match instructions that store to a watchpointed address.
0b11	Match instructions that load from or store to a watchpointed address.

All other values are reserved, but must behave as if the watchpoint is disabled. Software must not rely on this property as the behavior of reserved values might change in a future revision of the architecture.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

PAC, bits [2:1]

Privilege of access control. Determines the Exception level or levels at which a Watchpoint debug event for watchpoint n is generated.

The fields that indicate when the watchpoint can be generated are: HMC, PAC, and SSC. These fields must be considered in combination, and the values that are permitted for these fields are constrained.

For more information on the operation of these fields, see 'Execution conditions for which a watchpoint generates Watchpoint exceptions'.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E, bit [0]

Enable watchpoint n.

E	Meaning
0b0	Watchpoint n disabled.
0b1	Watchpoint n enabled.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing DBGWCR<n>_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, DBGWCR<m>_EL1 ; Where m = 0-15
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	m[3:0]	0b111

```
integer m = UInt(CRm<3:0>);

if m >= NUM_WATCHPOINTS then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif OSLSR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X[t, 64] = DBGWCR_EL1[m];
elsif PSTATE.EL == EL2 then
    if OSLSR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X[t, 64] = DBGWCR_EL1[m];
```

```
MSR DBGWCR<m>_EL1, <Xt> ; Where m = 0-15
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	m[3:0]	0b111

```
integer m = UInt(CRm<3:0>);

if m >= NUM_WATCHPOINTS then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif OSLSR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGWCR_EL1[m] = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
elsif PSTATE.EL == EL2 then
    if OSLSR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGWCR_EL1[m] = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.3.11 DBGWVR<n>_EL1, Debug Watchpoint Value Registers, n = 0 - 63

The DBGWVR<n>_EL1 characteristics are:

Purpose

Holds a data address value for use in watchpoint matching. Forms watchpoint n together with control register DBGWCR<n>_EL1.

Configuration

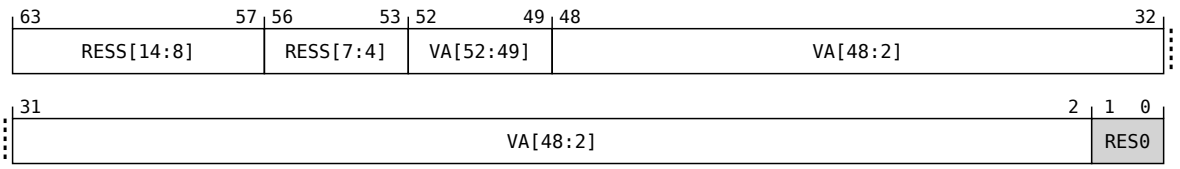
If watchpoint n is not implemented then accesses to this register are UNDEFINED.

AArch64 System register DBGWVR<n>_EL1 bits [63:0] are architecturally mapped to External register DBGWVR<n>_EL1[63:0].

Attributes

DBGWVR<n>_EL1 is a 64-bit register.

Field descriptions



RESS[14:8], bits [63:57]

Reserved, Sign extended. Software must set all bits in this field to the same value as the most significant bit of the VA field. If all bits in this field are not the same value as the most significant bit of the VA field, then all of the following apply:

- It is CONSTRAINED UNPREDICTABLE whether the PE ignores this field when comparing an address.
- It is IMPLEMENTATION DEFINED whether the value read back in each bit of this field is a copy of the most significant bit of the VA field or the value written.

RESS[7:4], bits [56:53]

Extension to RESS[14:8]. For more information, see RESS[14:8].

Bits [52:49]

When FEAT_LVA is implemented:

VA[52:49], bits [52:49]

Extension to VA[48:2]. For more information, see VA[48:2].

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RESS[3:0], bits [52:49]

Extension to RESS[14:8]. For more information, see RESS[14:8].

VA[48:2], bits [48:2]

Bits[48:2] of the address value for comparison.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_LVA is implemented, VA[52:49] forms the upper part of the address value. If FEAT_LVA is not implemented, bits [52:49] are part of the RESS field.

Arm deprecates setting DBGWVR<n>_EL1[2] == 1.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [1:0]

Reserved, RES0.

Accessing DBGWVR<n>_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, DBGWVR<m>_EL1 ; Where m = 0-15
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	m[3:0]	0b110

```
integer m = UInt(CRm<3:0>);

if m >= NUM_WATCHPOINTS then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif OSLSR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X[t, 64] = DBGWVR_EL1[m];
elsif PSTATE.EL == EL2 then
    if OSLSR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        X[t, 64] = DBGWVR_EL1[m];
```

```
MSR DBGWVR<m>_EL1, <Xt> ; Where m = 0-15
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	m[3:0]	0b110

```
integer m = UInt(CRm<3:0>);

if m >= NUM_WATCHPOINTS then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif OSLSR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
    else
        DBGWVR_EL1[m] = X[t, 64];
elsif PSTATE.EL == EL2 then
    if OSLSR_EL1.OSLK == '0' && HaltingAllowed() && EDSCR.TDA == '1' then
        Halt(DebugHalt_SoftwareAccess);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        DBGWVR_EL1[m] = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.3.12 DLR_EL0, Debug Link Register

The DLR_EL0 characteristics are:

Purpose

In Debug state, holds the address to restart from.

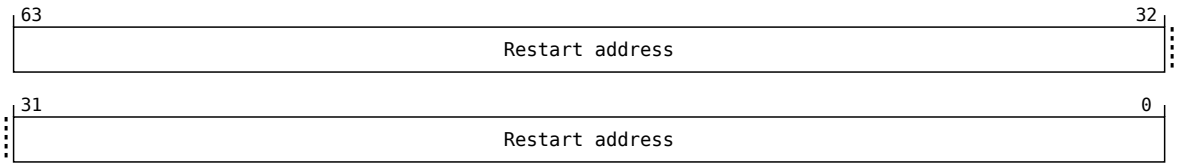
Configuration

There are no configuration notes.

Attributes

DLR_EL0 is a 64-bit register.

Field descriptions



ADDR, bits [63:0]

Restart address.

Accessing DLR_EL0

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, DLR_EL0`

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0101	0b001

```
if !Halted() then
    UNDEFINED;
else
    X[t, 64] = DLR_EL0;
```

`MSR DLR_EL0, <Xt>`

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0101	0b001

```
if !Halted() then
    UNDEFINED;
else
    DLR_EL0 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.3.13 DSPSR_EL0, Debug Saved Program Status Register

The DSPSR_EL0 characteristics are:

Purpose

Holds the saved process state for Debug state. On entering Debug state, PSTATE information is written to this register. On exiting Debug state, values are copied from this register to PSTATE.

Configuration

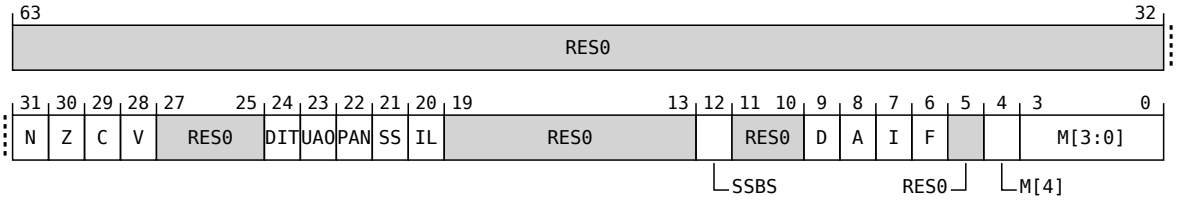
There are no configuration notes.

Attributes

DSPSR_EL0 is a 64-bit register.

Field descriptions

When entering or exiting Debug state from or to AArch64 state:



Bits [63:32]

Reserved, RES0.

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on entering Debug state, and copied to PSTATE.N on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on entering Debug state, and copied to PSTATE.Z on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on entering Debug state, and copied to PSTATE.C on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on entering Debug state, and copied to PSTATE.V on exiting Debug state.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [27:25]

Reserved, RES0.

DIT, bit [24]

When FEAT_DIT is implemented:

Data Independent Timing. Set to the value of PSTATE.DIT on entering Debug state, and copied to PSTATE.DIT on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

UAO, bit [23]

When FEAT_UAO is implemented:

User Access Override. Set to the value of PSTATE.UAO on entering Debug state, and copied to PSTATE.UAO on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

PAN, bit [22]

When FEAT_PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on entering Debug state, and copied to PSTATE.PAN on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SS, bit [21]

Software Step. Set to the value of PSTATE.SS on entering Debug state, and conditionally copied to PSTATE.SS on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on entering Debug state, and copied to PSTATE.IL on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [19:13]

Reserved, RES0.

SSBS, bit [12]

When FEAT_SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on entering Debug state, and copied to PSTATE.SSBS on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [11:10]

Reserved, RES0.

D, bit [9]

Debug exception mask. Set to the value of PSTATE.D on entering Debug state, and copied to PSTATE.D on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

A, bit [8]

SError exception mask. Set to the value of PSTATE.A on entering Debug state, and copied to PSTATE.A on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on entering Debug state, and copied to PSTATE.I on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on entering Debug state, and copied to PSTATE.F on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [5]

Reserved, RES0.

M[4], bit [4]

Execution state. Set to `0b0`, the value of PSTATE.nRW, on entering Debug state from AArch64 state, and copied to PSTATE.nRW on exiting Debug state.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

M[4]	Meaning
0b0	AArch64 execution state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

M[3:0], bits [3:0]

AArch64 Exception level and selected Stack Pointer.

M[3:0]	Meaning
0b0000	EL0.
0b0100	EL1 with SP_EL0 (EL1t).
0b0101	EL1 with SP_EL1 (EL1h).
0b1000	EL2 with SP_EL0 (EL2t).
0b1001	EL2 with SP_EL2 (EL2h).

Other values are reserved. If DSPSR_EL0.M[3:0] has a Reserved value, or a value for an unimplemented Exception level, exiting Debug state is an illegal return event, as described in 'Illegal return events from AArch64 state'.

The bits in this field are interpreted as follows:

- M[3:2] is set to the value of PSTATE.EL on entering Debug state and copied to PSTATE.EL on exiting Debug state.
- M[1] is unused and is 0 for all non-reserved values.
- M[0] is set to the value of PSTATE.SP on entering Debug state and copied to PSTATE.SP on exiting Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing DSPSR_EL0

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, DSPSR_EL0`

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0101	0b000

```
if !Halted() then
    UNDEFINED;
else
    X[t, 64] = DSPSR_EL0;
```

`MSR DSPSR_EL0, <Xt>`

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0101	0b000

```
if !Halted() then
    UNDEFINED;
else
    DSPSR_EL0 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.3.14 MDCCINT_EL1, Monitor DCC Interrupt Enable Register

The MDCCINT_EL1 characteristics are:

Purpose

Enables interrupt requests to be signaled based on the DCC status flags.

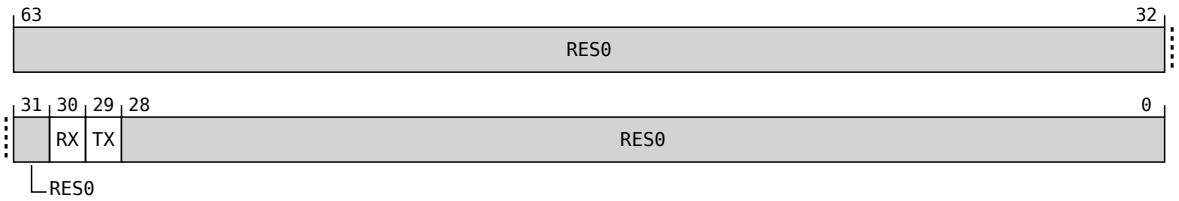
Configuration

There are no configuration notes.

Attributes

MDCCINT_EL1 is a 64-bit register.

Field descriptions



Bits [63:31]

Reserved, RES0.

RX, bit [30]

DCC interrupt request enable control for DTRRX. Enables a common **COMMIRQ** interrupt request to be signaled based on the DCC status flags.

RX	Meaning
0b0	No interrupt request generated by DTRRX.
0b1	Interrupt request will be generated on RXfull == 1.

If legacy **COMMRX** and **COMMTX** signals are implemented, then these are not affected by the value of this bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

TX, bit [29]

DCC interrupt request enable control for DTRTX. Enables a common **COMMIRQ** interrupt request to be signaled based on the DCC status flags.

TX	Meaning
0b0	No interrupt request generated by DTRTX.
0b1	Interrupt request will be generated on TXfull == 0.

If legacy **COMMRX** and **COMMTX** signals are implemented, then these are not affected by the value of this bit.

The reset behavior of this field is:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to '0'.

Bits [28:0]

Reserved, RES0.

Accessing MDCCINT_EL1

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, MDCCINT_EL1

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif Halted() && ConstrainUnpredictableBool(Unpredictable_IGNORETRAPINDEBUG) then
    X[t, 64] = MDCCINT_EL1;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = MDCCINT_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = MDCCINT_EL1;
```

MSR MDCCINT_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	0b0010	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif Halted() && ConstrainUnpredictableBool(Unpredictable_IGNORETRAPINDEBUG) then
    MDCCINT_EL1 = X[t, 64];
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        MDCCINT_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    MDCCINT_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.3.15 MDCCSR_EL0, Monitor DCC Status Register

The MDCCSR_EL0 characteristics are:

Purpose

Read-only register containing control status flags for the DCC.

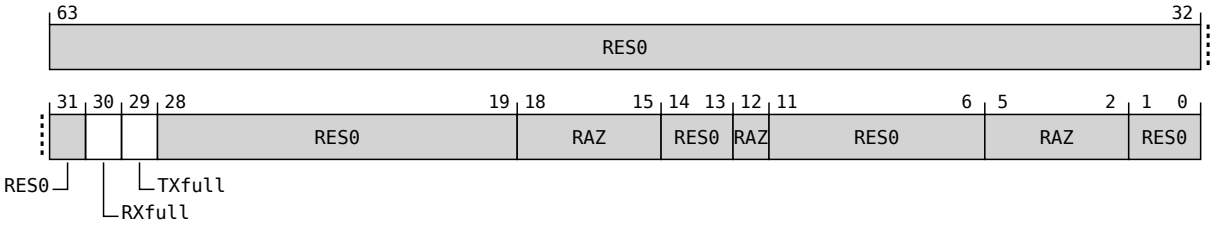
Configuration

AArch64 System register MDCCSR_EL0 bits [30:29] are architecturally mapped to External register EDSCR[30:29].

Attributes

MDCCSR_EL0 is a 64-bit register.

Field descriptions



Bits [63:31]

Reserved, RES0.

RXfull, bit [30]

DTRRX full. Read-only view of the equivalent bit in the EDSCR.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

TXfull, bit [29]

DTRTX full. Read-only view of the equivalent bit in the EDSCR.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Bits [28:19]

Reserved, RES0.

Bits [18:15]

Reserved, RAZ.

Bits [14:13]

Reserved, RES0.

Bit [12]

Reserved, RAZ.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [11:6]

Reserved, RES0.

Bits [5:2]

Reserved, RAZ.

Bits [1:0]

Reserved, RES0.

Accessing MDCCSR_EL0

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, MDCCSR_EL0`

op0	op1	CRn	CRm	op2
0b10	0b011	0b0000	0b0001	0b000

```
if Halted() && ConstrainUnpredictableBool(Unpredictable_IGNORETRAPINDEBUG) then
    X[t, 64] = MDCCSR_EL0;
elsif PSTATE.EL == EL0 then
    if MDSCR_EL1.TDCC == '1' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif HCR_EL2.TGE == '1' || MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = MDCCSR_EL0;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = MDCCSR_EL0;
elsif PSTATE.EL == EL2 then
    X[t, 64] = MDCCSR_EL0;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.3.16 MDCR_EL2, Monitor Debug Configuration Register (EL2)

The MDCR_EL2 characteristics are:

Purpose

Provides EL2 configuration options for self-hosted debug and the Performance Monitors Extension.

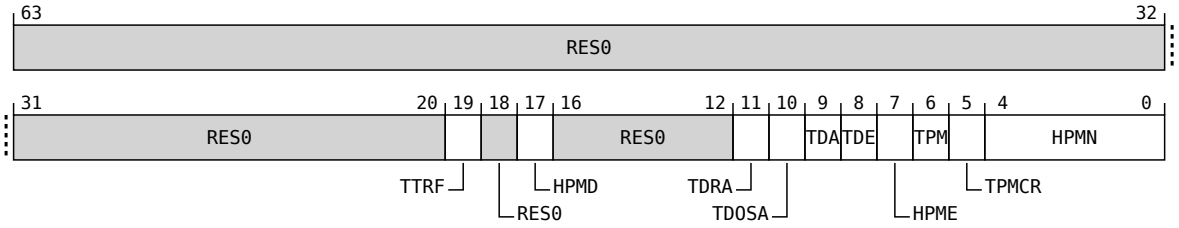
Configuration

There are no configuration notes.

Attributes

MDCR_EL2 is a 64-bit register.

Field descriptions



Bits [63:20]

Reserved, RES0.

TTRF, bit [19]

When FEAT_TRF is implemented:

Traps use of the Trace Filter Control registers at EL1 to EL2, as follows:

- Access to TRFCR_EL1 is trapped to EL2, reported using EC syndrome value `0x18`.

TTRF	Meaning
`0b0`	Accesses to the specified registers at EL1 are not affected by this control.
`0b1`	Accesses to the specified registers at EL1 generate a trap exception to EL2 when EL2 is enabled in the current Security state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [18]

Reserved, RES0.

HPMD, bit [17]

When FEAT_PMUv3p1 is implemented and FEAT_Debugv8p2 is implemented:

Guest Performance Monitors Disable. Controls PMU operation at EL2.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

HPMD	Meaning
0b0	Counters are not affected by this mechanism.
0b1	Affected counters are prohibited from counting at EL2. If PMCR_EL0.DP is 1, then PMCCNTR_EL0 is disabled at EL2. Otherwise, PMCCNTR_EL0 is not affected by this mechanism.

The counters affected by this field are:

- Event counters in the first range.
- If PMCR_EL0.DP is 1, the cycle counter PMCCNTR_EL0.

Other event counters are not affected by this field.

When PMCR_EL0.DP is 0, PMCCNTR_EL0 is not affected by this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

When FEAT_PMUv3p1 is implemented:

Guest Performance Monitors Disable. Controls PMU operation at EL2 when ExternalSecureNoninvasiveDebugEnabled() is FALSE.

HPMD	Meaning
0b0	Counters are not affected by this mechanism.
0b1	If ExternalSecureNoninvasiveDebugEnabled() is FALSE, then all the following apply: • Affected event counters are prohibited from counting at EL2. • If PMCR_EL0.DP is 1, then PMCCNTR_EL0 is disabled at EL2. Otherwise, PMCCNTR_EL0 is not affected by this mechanism.

If ExternalSecureNoninvasiveDebugEnabled() is TRUE, then the event counters and PMCCNTR_EL0 are not affected by this field.

Otherwise, the counters affected by this field are:

- Event counters in the first range.
- If PMCR_EL0.DP is 1, the cycle counter, PMCCNTR_EL0.

Other event counters are not affected by this field. When PMCR_EL0.DP is 0, PMCCNTR_EL0 is not affected by this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Otherwise:

Reserved, RES0.

Bits [16:12]

Reserved, RES0.

TDRA, bit [11]

Trap Debug ROM Address register access. Traps System register accesses to the Debug ROM registers to EL2 when EL2 is enabled in the current Security state as follows:

- If EL1 is using AArch64, accesses to MDRAR_EL1 are trapped to EL2, reported using EC syndrome value 0x18.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TDRA	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	EL0 and EL1 System register accesses to the Debug ROM registers are trapped to EL2 when EL2 is enabled in the current Security state, unless it is trapped by the following: • MDSCR_EL1.TDCC.

This field is treated as being 1 for all purposes other than a direct read when one or more of the following are true:

- MDCR_EL2.TDE == 1.
- HCR_EL2.TGE == 1.

Note

EL2 does not provide traps on debug register accesses through the optional memory-mapped external debug interfaces.

System register accesses to the debug registers might have side-effects. When a System register access is trapped to EL2, no side-effects occur before the exception is taken to EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TDOSA, bit [10]

When FEAT_DoubleLock is implemented:

Trap debug OS-related register access. Traps EL1 System register accesses to the powerdown debug registers to EL2, from both Execution states as follows:

- In AArch64 state, accesses to the following registers are trapped to EL2, reported using EC syndrome value 0x18:
  - OSLAR_EL1, OSLSR_EL1, OSDLR_EL1, and DBGPRCR_EL1.
  - Any IMPLEMENTATION DEFINED register with similar functionality that the implementation specifies as trapped by this bit.

TDOSA	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	EL1 System register accesses to the powerdown debug registers are trapped to EL2 when EL2 is enabled in the current Security state.

Note

These registers are not accessible at EL0.

This field is treated as being 1 for all purposes other than a direct read when one or more of the following are true:

- MDCR_EL2.TDE == 1.
- HCR_EL2.TGE == 1.

System register accesses to the debug registers might have side-effects. When a System register access is trapped to EL2, no side-effects occur before the exception is taken to EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Trap debug OS-related register access. Traps EL1 System register accesses to the powerdown debug registers to EL2, from both Execution states as follows:

- In AArch64 state, accesses to the following registers are trapped to EL2, reported using EC syndrome value `0x18`:
  - OSLAR_EL1, OSLSR_EL1, and DBGPRCR_EL1.
  - Any IMPLEMENTATION DEFINED register with similar functionality that the implementation specifies as trapped by this bit.

It is IMPLEMENTATION DEFINED whether accesses to OSDLR_EL1 are trapped.

TDOSA	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	EL1 System register accesses to the powerdown debug registers are trapped to EL2 when EL2 is enabled in the current Security state.

Note

These registers are not accessible at EL0.

This field is treated as being 1 for all purposes other than a direct read when one or more of the following are true:

- MDCR_EL2.TDE == 1.
- HCR_EL2.TGE == 1.

Note

EL2 does not provide traps on debug register accesses through the optional memory-mapped external debug interfaces.

System register accesses to the debug registers might have side-effects. When a System register access is trapped to EL2, no side-effects occur before the exception is taken to EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TDA, bit [9]

Trap accesses of debug System registers. Enables a trap to EL2 on accesses of debug System registers.

TDA	Meaning
`0b0`	Accesses of the specified debug System registers are not trapped by this mechanism.
`0b1`	Accesses of the specified debug System registers at EL1 and EL0 are trapped to EL2, unless the instruction generates a higher priority exception.

In AArch64 state, the instructions affected by this control are:

- `MRS` and `MSR` accesses to DBGAUTHSTATUS_EL1, DBGBCR<n>_EL1, DBGBVR<n>_EL1, DBGCLAIMCLR_EL1, DBGCLAIMSET_EL1, DBGWCR<n>_EL1, DBGWVR<n>_EL1, MDCCINT_EL1, MDCCSR_EL0, MDSCR_EL1, OSDTRRX_EL1, OSDTRTX_EL1, and OSECCR_EL1.
- In Non-debug state, `MRS` accesses to DBGDTRRX_EL0 and DBGDTR_EL0 and `MSR` accesses to DBGDTRTX_EL0 and DBGDTR_EL0.

Trapped AArch64 instructions are reported using EC syndrome value `0x18`.

The following instructions are not trapped in Debug state:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- AArch64 MRS accesses to DBGDTRRX_EL0 and DBGDTR_EL0 and MSR accesses to DBGDTRTX_EL0 and DBGDTR_EL0.
- AArch32 MRC accesses to DBGDTRRXint and MCR accesses to DBGDTRTXint.

This field is ignored by the PE and treated as one when any of the following are true:

- MDCR_EL2.TDE == 1.
- HCR_EL2.TGE == 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TDE, bit [8]

Trap Debug Exceptions. Controls routing of Debug exceptions, and defines the debug target Exception level, EL_D.

TDE	Meaning
0b0	The debug target Exception level is EL1.
0b1	If EL2 is enabled for the current Effective value of SCR_EL3.NS, the debug target Exception level is EL2, otherwise the debug target Exception level is EL1. The MDCR_EL2.{TDRA, TDOSA, TDA} fields are treated as being 1 for all purposes other than returning the result of a direct read of the register.

For more information, see 'Routing debug exceptions'.

This field is treated as being 1 for all purposes other than a direct read when HCR_EL2.TGE == 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

HPME, bit [7]

When FEAT_PMUv3 is implemented:

Hyp Enable.

HPME	Meaning
0b0	Affected counters are disabled and do not count.
0b1	Affected counters are enabled by PMCNTENSET_EL0.

The counters affected by this field are the event counters in the second range.

The following counters are not affected by this field:

- Other event counters.
- PMCCNTR_EL0.

If MDCR_EL2.HPMN is equal to `GetNumEventCountersSelfHosted()`, then this field has no effect.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TPM, bit [6]

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_PMUv3 is implemented:

Trap accesses of PMU registers. Enables a trap to EL2 on accesses of PMU registers.

TPM	Meaning
0b0	Accesses of the specified PMU registers are not trapped by this mechanism.
0b1	Accesses of the specified PMU registers at EL1 and EL0 are trapped to EL2, unless the instruction generates a higher priority exception.

In AArch64 state, the instructions affected by this control are:

- MRS and MSR accesses to PMCCFILTR_EL0, PMCCNTR_EL0, PMCNTENCLR_EL0, PMCNTENSET_EL0, PMCR_EL0, PMEVCNTR<n>_EL0, PMEVTYPER<n>_EL0, PMINTENCLR_EL1, PMINTENSET_EL1, PMOVSCLR_EL0, PMOVSSET_EL0, PMSELR_EL0, PMSWINC_EL0, PMUSERENR_EL0, PMXEVCNTR_EL0, and PMXEVTYPER_EL0.
- MRS accesses to PMCEID0_EL0 and PMCEID1_EL0.
- If FEAT_PMUv3p4 is implemented, MRS accesses to PMMIR_EL1.

Trapped AArch64 instructions are reported using EC syndrome value 0x18.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TPMCR, bit [5]

When FEAT_PMUv3 is implemented:

Trap PMCR_EL0 or PMCR accesses. Traps EL0 and EL1 accesses to EL2, when EL2 is enabled in the current Security state, as follows:

- In AArch64 state, accesses to PMCR_EL0 are trapped to EL2, reported using EC syndrome value 0x18.

TPMCR	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	EL0 and EL1 accesses to the specified registers are trapped to EL2 when EL2 is enabled in the current Security state, unless they are trapped by the following: • PMUSERENR_EL0.EN.

Note

EL2 does not provide traps on Performance Monitor register accesses through the optional memory-mapped external debug interface.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HPMN, bits [4:0]

When FEAT_PMUv3 is implemented:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Defines the number of event counters PMEVCNTR<n>_EL0 that are accessible from EL1 and, if permitted, from EL0.

MDCR_EL2.HPMN divides the event counters into a first range and a second range.

If MDCR_EL2.HPMN is not 0 and is less than the number of PMU event counters implemented by the PE, NUM_PMU_COUNTERS, then event counters [0..(MDCR_EL2.HPMN-1)] are in the first range, and the remaining event counters [MDCR_EL2.HPMN..(NUM_PMU_COUNTERS-1)] are in the second range.

The pseudocode function GetNumEventCountersSelfHosted() returns the value NUM_PMU_COUNTERS.

If MDCR_EL2.HPMN is equal to GetNumEventCountersSelfHosted(), or EL2 is not implemented, then all of the following apply:

- All event counters are in the first range.
- No event counters are in the second range.

All of the following apply for an event counter PMEVCNTR<n>_EL0 in the first range:

- The counter is accessible from EL1 and EL2.
- The counter is accessible from EL0 if permitted by PMUSERENR_EL0 or PMUSERENR.
- PMCR_EL0.E and PMCNTENSET_EL0[n] enable the operation of the event counter.

All of the following apply for an event counter PMEVCNTR<n>_EL0 in the second range:

- The counter is only accessible from EL2.
- MDCR_EL2.HPME and PMCNTENSET_EL0[n] enable the operation of the event counter.

Values greater than GetNumEventCountersSelfHosted() and 0 are reserved.

If this field is set to a reserved value, then the following CONSTRAINED UNPREDICTABLE behaviors apply:

- The value returned by a direct read of MDCR_EL2.HPMN is UNKNOWN.
- The number of event counters in each of the first and second ranges is UNKNOWN. That is, either:
  - The PE behaves as if MDCR_EL2.HPMN is set to an UNKNOWN nonzero value less than or equal to NUM_PMU_COUNTERS.
  - All counters are in the second range and none are in the first range.

The reset behavior of this field is:

- On a Warm reset, this field resets to the expression NUM_PMU_COUNTERS.

Otherwise:

Reserved, RES0.

Accessing MDCR_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, MDCR_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = MDCR_EL2;
```

```
MSR MDCR_EL2, <Xt>
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    MDCR_EL2 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.3.17 MDRAR_EL1, Monitor Debug ROM Address Register

The MDRAR_EL1 characteristics are:

Purpose

Defines the base physical address of a 4KB-aligned memory-mapped debug component, usually a ROM table that locates and describes the memory-mapped debug components in the system. Use of this register is deprecated.

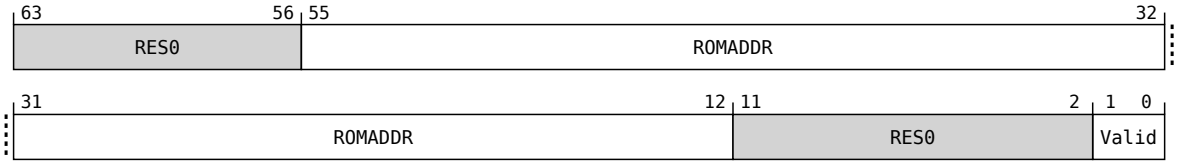
Configuration

There are no configuration notes.

Attributes

MDRAR_EL1 is a 64-bit register.

Field descriptions

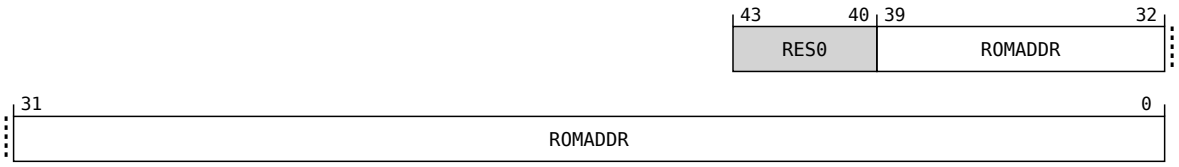


Bits [63:56]

Reserved, RES0.

ROMADDR, bits [55:12]

When FEAT_LPA is implemented and MDRAR_EL1.Valid != '00':



Bits [43:40]

Reserved, RES0.

ROMADDR, bits [39:0]

Bits [51:12] of the ROM table physical address.

Bits [11:0] of the ROM table physical address are zero.

This field has an IMPLEMENTATION DEFINED value.

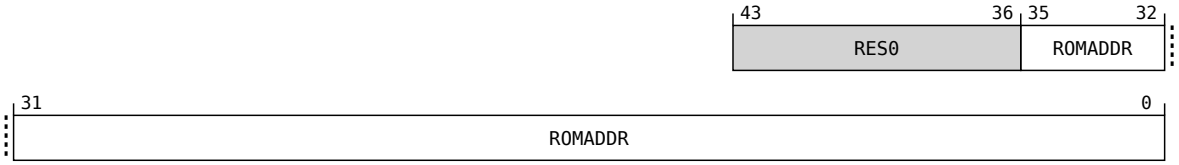
For implementations with fewer than 52 physical address bits, the corresponding upper bits of this field are RES0

In an implementation that includes EL3, ROMADDR is an address in Non-secure memory. It is IMPLEMENTATION DEFINED whether the ROM table is also accessible in Secure memory.

Access to this field is RO.

When FEAT_LPA is not implemented and MDRAR_EL1.Valid != '00':

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential



Bits [43:36]

Reserved, RES0.

ROMADDR, bits [35:0]

Bits [39:12] of the ROM table physical address.

Bits [11:0] of the ROM table physical address are zero.

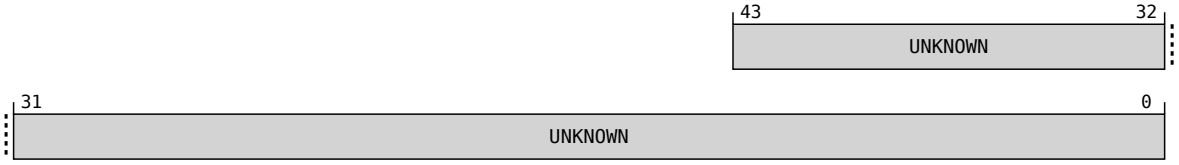
This field has an IMPLEMENTATION DEFINED value.

For implementations with fewer than 48 physical address bits, the corresponding upper bits of this field are RES0

In an implementation that includes EL3, ROMADDR is an address in Non-secure memory. It is IMPLEMENTATION DEFINED whether the ROM table is also accessible in Secure memory.

Access to this field is RO.

When MDRAR_EL1.Valid == '00':



Bits [43:0]

Reserved, UNKNOWN.

Bits [11:2]

Reserved, RES0.

Valid, bits [1:0]

This field indicates whether the ROM Table address is valid.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Valid	Meaning
0b00	ROM Table address is not valid. Software must ignore ROMADDR.
0b11	ROM Table address is valid.

Other values are reserved.

Arm recommends implementations set this field to zero.

Access to this field is RO.

Accessing MDRAR_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, MDRAR_EL1`

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b10	0b000	0b0001	0b0000	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.<TDE,TDRA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = MDRAR_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = MDRAR_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.3.18 MDSCR_EL1, Monitor Debug System Control Register

The MDSCR_EL1 characteristics are:

Purpose

Main control register for the debug implementation.

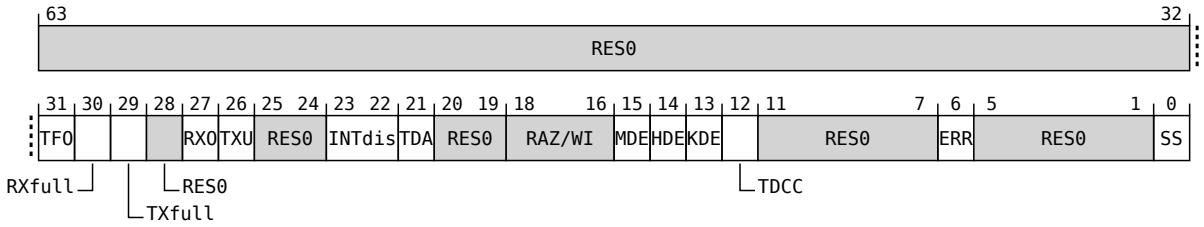
Configuration

AArch64 System register MDSCR_EL1 bits [31:29, 27:26, 23:21, 19, 14, 6] are architecturally mapped to External register EDSCR[31:29, 27:26, 23:21, 19, 14, 6].

Attributes

MDSCR_EL1 is a 64-bit register.

Field descriptions



Bits [63:32]

Reserved, RES0.

TFO, bit [31]

When FEAT_TRF is implemented:

Trace Filter override. Used for save/restore of EDSCR.TFO.

When OSLSR_EL1.OSLK == 0, software must treat this bit as UNK/SBZP.

When OSLSR_EL1.OSLK == 1, this bit holds the value of EDSCR.TFO. Reads and writes of this bit are indirect accesses to EDSCR.TFO.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- When OSLSR_EL1.OSLK == '1', access to this field is RW.
- When OSLSR_EL1.OSLK == '0', access to this field is RO.

Otherwise:

Reserved, RES0.

RXfull, bit [30]

Used for save/restore of EDSCR.RXfull.

When OSLSR_EL1.OSLK == 0, software must treat this bit as UNK/SBZP.

When OSLSR_EL1.OSLK == 1, this bit holds the value of EDSCR.RXfull. Reads and writes of this bit are indirect accesses to EDSCR.RXfull.

The reset behavior of this field is:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Cold reset, this field resets to '0'.

Accessing this field has the following behavior:

- When OSLSR_EL1.OSLK == '1', access to this field is RW.
- When OSLSR_EL1.OSLK == '0', access to this field is RO.

TXfull, bit [29]

Used for save/restore of EDSCR.TXfull.

When OSLSR_EL1.OSLK == 0, software must treat this bit as UNK/SBZP.

When OSLSR_EL1.OSLK == 1, this bit holds the value of EDSCR.TXfull. Reads and writes of this bit are indirect accesses to EDSCR.TXfull.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Accessing this field has the following behavior:

- When OSLSR_EL1.OSLK == '1', access to this field is RW.
- When OSLSR_EL1.OSLK == '0', access to this field is RO.

Bit [28]

Reserved, RES0.

RXO, bit [27]

Used for save/restore of EDSCR.RXO.

When OSLSR_EL1.OSLK == 0, software must treat this bit as UNK/SBZP.

When OSLSR_EL1.OSLK == 1, this bit holds the value of EDSCR.RXO. Reads and writes of this bit are indirect accesses to EDSCR.RXO.

When OSLSR_EL1.OSLK == 1, if bits [27,6] of the value written to MDSCR_EL1 are {1,0}, that is, the RXO bit is 1 and the ERR bit is 0, the PE sets EDSCR.{RXO,ERR} to UNKNOWN values.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Accessing this field has the following behavior:

- When OSLSR_EL1.OSLK == '1', access to this field is RW.
- When OSLSR_EL1.OSLK == '0', access to this field is RO.

TXU, bit [26]

Used for save/restore of EDSCR.TXU.

When OSLSR_EL1.OSLK == 0, software must treat this bit as UNK/SBZP.

When OSLSR_EL1.OSLK == 1, this bit holds the value of EDSCR.TXU. Reads and writes of this bit are indirect accesses to EDSCR.TXU.

When OSLSR_EL1.OSLK == 1, if bits [26,6] of the value written to MDSCR_EL1 are {1,0}, that is, the TXU bit is 1 and the ERR bit is 0, the PE sets EDSCR.{TXU,ERR} to UNKNOWN values.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Accessing this field has the following behavior:

- When OSLSR_EL1.OSLK == '1', access to this field is RW.
- When OSLSR_EL1.OSLK == '0', access to this field is RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [25:24]

Reserved, RES0.

INTdis, bits [23:22]

Used for save/restore of EDSCR.INTdis.

When OSLSR_EL1.OSLK == 0, software must treat this bit as UNK/SBZP.

When OSLSR_EL1.OSLK == 1, this field holds the value of EDSCR.INTdis. Reads and writes of this field are indirect accesses to EDSCR.INTdis.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'00'`.

Accessing this field has the following behavior:

- When OSLSR_EL1.OSLK == '1', access to this field is RW.
- When OSLSR_EL1.OSLK == '0', access to this field is RO.

TDA, bit [21]

Used for save/restore of EDSCR.TDA.

When OSLSR_EL1.OSLK == 0, software must treat this bit as UNK/SBZP.

When OSLSR_EL1.OSLK == 1, this bit holds the value of EDSCR.TDA. Reads and writes of this bit are indirect accesses to EDSCR.TDA.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- When OSLSR_EL1.OSLK == '1', access to this field is RW.
- When OSLSR_EL1.OSLK == '0', access to this field is RO.

Bits [20:19]

Reserved, RES0.

Bits [18:16]

Reserved, RAZ/WI.

Hardware must implement this field as RAZ/WI. Software must not rely on the register reading as zero, and must use a read-modify-write sequence to write to the register.

MDE, bit [15]

Monitor debug events. Enable Breakpoint, Watchpoint, and Vector Catch exceptions.

MDE	Meaning
`0b0`	Breakpoint, Watchpoint, and Vector Catch exceptions disabled.
`0b1`	Breakpoint, Watchpoint, and Vector Catch exceptions enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

HDE, bit [14]

Used for save/restore of EDSCR.HDE.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When OSLSR_EL1.OSLK == 0, software must treat this bit as UNK/SBZP.

When OSLSR_EL1.OSLK == 1, this bit holds the value of EDSCR.HDE. Reads and writes of this bit are indirect accesses to EDSCR.HDE.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- When OSLSR_EL1.OSLK == '1', access to this field is RW.
- When OSLSR_EL1.OSLK == '0', access to this field is RO.

KDE, bit [13]

Local (kernel) debug enable. If EL_D is using AArch64, enable debug exceptions within EL_D. Permitted values are:

KDE	Meaning
`0b0`	Debug exceptions, other than Breakpoint Instruction exceptions, disabled within EL_D.
`0b1`	All debug exceptions enabled within EL_D.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

TDCC, bit [12]

Traps EL0 accesses to the Debug Communication Channel (DCC) registers to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, from both Execution states, as follows:

- In AArch64 state, MRS or MSR accesses to the following DCC registers are trapped, reported using EC syndrome value `0x18`:
  - MDCCSR_EL0.
  - If not in Debug state, DBGDTR_EL0, DBGDTRTX_EL0, and DBGDTRRX_EL0.

TDCC	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	EL0 using AArch64: EL0 accesses to the AArch64 DCC registers are trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [11:7]

Reserved, RES0.

ERR, bit [6]

Used for save/restore of EDSCR.ERR.

When OSLSR_EL1.OSLK == 0, software must treat this bit as UNK/SBZP.

When OSLSR_EL1.OSLK == 1, this bit holds the value of EDSCR.ERR. Reads and writes of this bit are indirect accesses to EDSCR.ERR.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing this field has the following behavior:

- When OSLSR_EL1.OSLK == ‘1’, access to this field is RW.
- When OSLSR_EL1.OSLK == ‘0’, access to this field is RO.

Bits [5:1]

Reserved, RES0.

SS, bit [0]

Software step control bit. If EL_D is using AArch64, enable Software step. Permitted values are:

SS	Meaning
0b0	Software step disabled
0b1	Software step enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing MDSCR_EL1

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, MDSCR_EL1

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	0b0010	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = MDSCR_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = MDSCR_EL1;
```

MSR MDSCR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	0b0010	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        MDSCR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    MDSCR_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.3.19 OSDLR_EL1, OS Double Lock Register

The OSDLR_EL1 characteristics are:

Purpose

Used to control the OS Double Lock.

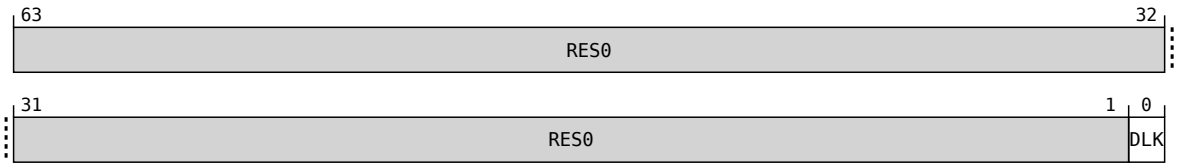
Configuration

There are no configuration notes.

Attributes

OSDLR_EL1 is a 64-bit register.

Field descriptions



Bits [63:1]

Reserved, RES0.

DLK, bit [0]

When FEAT_DoubleLock is implemented:

OS Double Lock control bit.

DLK	Meaning
0b0	OS Double Lock unlocked.
0b1	OS Double Lock locked, if DBGPRCR_EL1.CORENPDRQ (Core no powerdown request) bit is set to 0 and the PE is in Non-debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Otherwise:

Reserved, RAZ/WI.

Accessing OSDLR_EL1

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, OSDLR_EL1

op0	op1	CRn	CRm	op2
0b10	0b000	0b0001	0b0011	0b100

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.<TDE,TDOSA> != '00' && (IsFeatureImplemented(FEAT_DoubleLock) || boolean
        ↪IMPLEMENTATION_DEFINED "Trapped by MDCR_EL2.TDOSA") then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = OSDLR_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = OSDLR_EL1;
```

MSR OSDLR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b10	0b000	0b0001	0b0011	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.<TDE,TDOSA> != '00' && (IsFeatureImplemented(FEAT_DoubleLock) || boolean
        ↪IMPLEMENTATION_DEFINED "Trapped by MDCR_EL2.TDOSA") then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        OSDLR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    OSDLR_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.3.20 OSDTRRX_EL1, OS Lock Data Transfer Register, Receive

The OSDTRRX_EL1 characteristics are:

Purpose

Used for save and restore of DBGDTRRX_EL0. It is a component of the Debug Communications Channel.

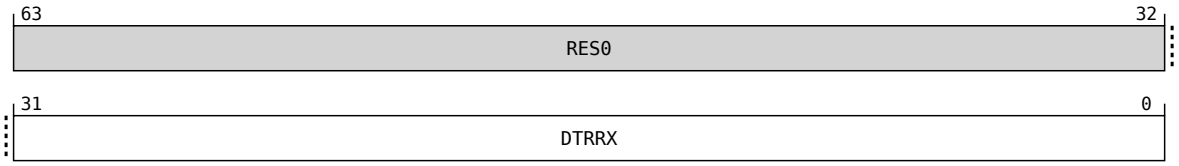
Configuration

There are no configuration notes.

Attributes

OSDTRRX_EL1 is a 64-bit register.

Field descriptions



Bits [63:32]

Reserved, RES0.

DTRRX, bits [31:0]

Update DTRRX without side-effect.

Writes to this register update the value in DTRRX and do not change RXfull.

Reads of this register return the last value written to DTRRX and do not change RXfull.

For the full behavior of the Debug Communications Channel, see 'The Debug Communication Channel and Instruction Transfer Register'.

Accessing OSDTRRX_EL1

Arm deprecates reads and writes of OSDTRRX_EL1 when the OS Lock is unlocked.

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, OSDTRRX_EL1`

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif Halted() && ConstrainUnpredictableBool(Unpredictable_IGNORETRAPINDEBUG) then
    X[t, 64] = OSDTRRX_EL1;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = OSDTRRX_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = OSDTRRX_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR OSDTRRX_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	0b0000	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif Halted() && ConstrainUnpredictableBool(Unpredictable_IGNORETRAPINDEBUG) then
    OSDTRRX_EL1 = X[t, 64];
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        OSDTRRX_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    OSDTRRX_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.3.21 OSDTRTX_EL1, OS Lock Data Transfer Register, Transmit

The OSDTRTX_EL1 characteristics are:

Purpose

Used for save/restore of DBGDTRTX_EL0. It is a component of the Debug Communications Channel.

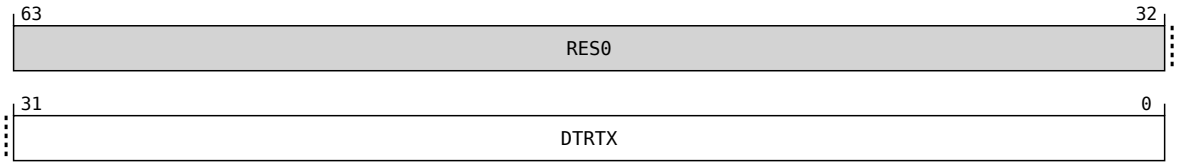
Configuration

There are no configuration notes.

Attributes

OSDTRTX_EL1 is a 64-bit register.

Field descriptions



Bits [63:32]

Reserved, RES0.

DTRTX, bits [31:0]

Return DTRTX without side-effect.

Reads of this register return the value in DTRTX and do not change TXfull.

Writes of this register update the value in DTRTX and do not change TXfull.

For the full behavior of the Debug Communications Channel, see 'The Debug Communication Channel and Instruction Transfer Register'.

Accessing OSDTRTX_EL1

Arm deprecates reads and writes of OSDTRTX_EL1 when the OS Lock is unlocked.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, OSDTRTX_EL1
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	0b0011	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif Halted() && ConstrainUnpredictableBool(Unpredictable_IGNORETRAPINDEBUG) then
    X[t, 64] = OSDTRTX_EL1;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = OSDTRTX_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = OSDTRTX_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR OSDTRTX_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	0b0011	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif Halted() && ConstrainUnpredictableBool(Unpredictable_IGNORETRAPINDEBUG) then
    OSDTRTX_EL1 = X[t, 64];
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        OSDTRTX_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    OSDTRTX_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.3.22 OSECCR_EL1, OS Lock Exception Catch Control Register

The OSECCR_EL1 characteristics are:

Purpose

Provides a mechanism for an operating system to access the contents of EDECCR that are otherwise invisible to software, so it can save/restore the contents of EDECCR over powerdown on behalf of the external debugger.

Configuration

If OSLSR_EL1.OSLK == 0, then OSECCR_EL1 returns an UNKNOWN value on reads and ignores writes.

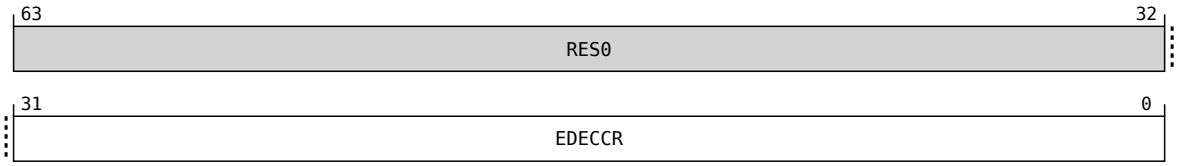
AArch64 System register OSECCR_EL1 bits [31:0] are architecturally mapped to External register EDECCR[31:0].

Attributes

OSECCR_EL1 is a 64-bit register.

Field descriptions

When OSLSR_EL1.OSLK == '1':



Bits [63:32]

Reserved, RES0.

EDECCR, bits [31:0]

Used for save/restore to EDECCR over powerdown.

Reads or writes to this field are indirect accesses to EDECCR.

The reset behavior of this field is:

- On a Cold reset, this field resets to 0x00000000.

Accessing OSECCR_EL1

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, OSECCR_EL1

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	0b0110	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif OSLSR_EL1.OSLK == '0' then
        X[t, 64] = bits(64) UNKNOWN;
    else
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        X[t, 64] = OSECCR_EL1;
elsif PSTATE.EL == EL2 then
    if OSLSR_EL1.OSLK == '0' then
        X[t, 64] = bits(64) UNKNOWN;
    else
        X[t, 64] = OSECCR_EL1;
```

MSR OSECCR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b10	0b000	0b0000	0b0110	0b010

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif OSLSR_EL1.OSLK == '0' then
        return;
    else
        OSECCR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    if OSLSR_EL1.OSLK == '0' then
        return;
    else
        OSECCR_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.3.23 OSLAR_EL1, OS Lock Access Register

The OSLAR_EL1 characteristics are:

Purpose

Used to lock or unlock the OS Lock.

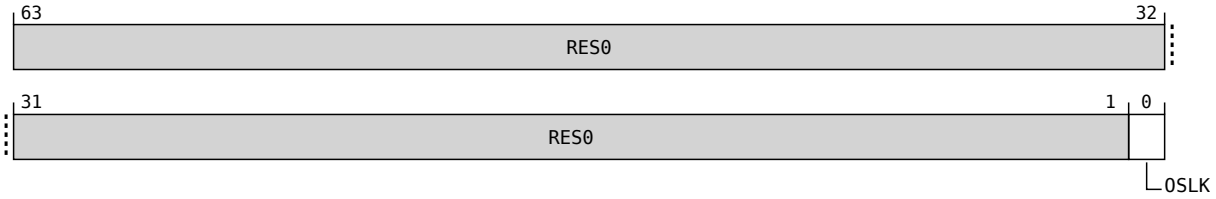
Configuration

AArch64 System register OSLAR_EL1 bits [31:0] are architecturally mapped to External register OSLAR_EL1[31:0].

Attributes

OSLAR_EL1 is a 64-bit register.

Field descriptions



Bits [63:1]

Reserved, RES0.

OSLK, bit [0]

On writes to OSLAR_EL1, bit[0] is copied to the OS Lock.

Use OSLSR_EL1.OSLK to check the current status of the lock.

Accessing OSLAR_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MSR OSLAR_EL1, <Xt>`

op0	op1	CRn	CRm	op2
0b10	0b000	0b0001	0b0000	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.<TDE,TDOSA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        OSLAR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    OSLAR_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.3.24 OSLSR_EL1, OS Lock Status Register

The OSLSR_EL1 characteristics are:

Purpose

Provides the status of the OS Lock.

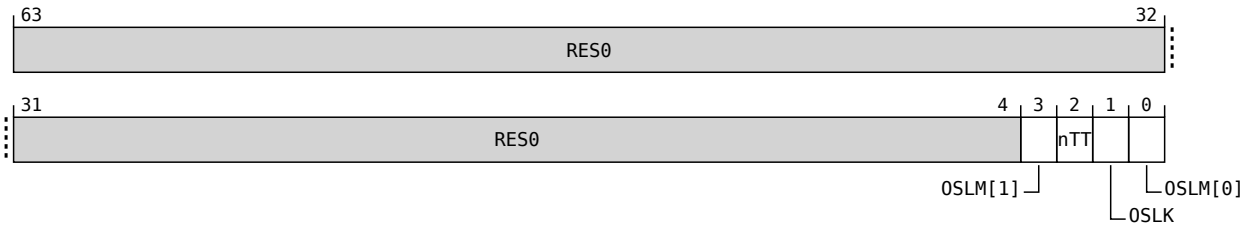
Configuration

There are no configuration notes.

Attributes

OSLSR_EL1 is a 64-bit register.

Field descriptions



Bits [63:4]

Reserved, RES0.

OSLM, bits [3, 0]

OS Lock model implemented. Identifies the form of OS save and restore mechanism implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

OSLM	Meaning
0b00	OS Lock not implemented.
0b10	OS Lock implemented.

All other values are reserved. In an Armv8 implementation the value 0b00 is not permitted.

Access to this field is RO.

nTT, bit [2]

Not 32-bit access. This bit is always RAZ. It indicates that a 32-bit access is needed to write the key to the OS Lock Access Register.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

OSLK, bit [1]

OS Lock Status.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

OSLK	Meaning
0b0	OS Lock unlocked.
0b1	OS Lock locked.

The OS Lock is locked and unlocked by writing to the OS Lock Access Register.

The reset behavior of this field is:

- On a Cold reset, this field resets to '1'.

Accessing OSLSR_EL1

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, OSLSR_EL1

op0	op1	CRn	CRm	op2
0b10	0b000	0b0001	0b0001	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.<TDE,TDOSA> != '00' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = OSLSR_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = OSLSR_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.4 Trace registers

This section lists the Trace registers in AArch64.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.4.1 TRFCR_EL1, Trace Filter Control Register (EL1)

The TRFCR_EL1 characteristics are:

Purpose

Provides EL1 controls for Trace.

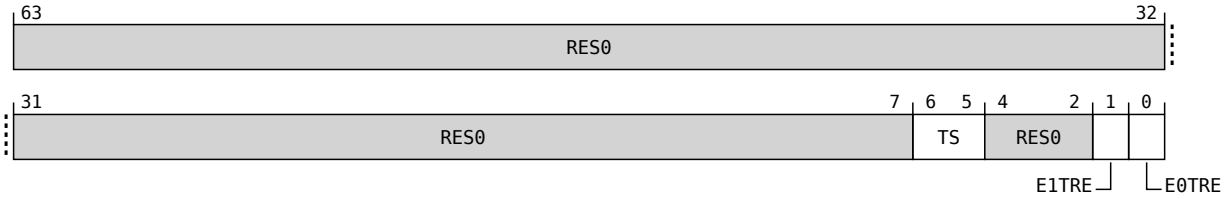
Configuration

This register is present only when FEAT_TRF is implemented. Otherwise, direct accesses to TRFCR_EL1 are UNDEFINED.

Attributes

TRFCR_EL1 is a 64-bit register.

Field descriptions



Bits [63:7]

Reserved, RES0.

TS, bits [6:5]

Timestamp Control. Controls which timebase is used for trace timestamps.

TS	Meaning
0b01	Virtual timestamp. The traced timestamp is the physical counter value minus the value of CNTVOFF_EL2.
0b11	Physical timestamp. The traced timestamp is the physical counter value.

All other values are reserved.

If any of the following are true, then this field is ignored by the PE:

- EL2 is implemented and TRFCR_EL2.TS is not 0b00.
- SelfHostedTraceEnabled() == FALSE.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [4:2]

Reserved, RES0.

E1TRE, bit [1]

EL1 Trace Enable.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E1TRE	Meaning
0b0	Trace is prohibited at EL1.
0b1	Trace is allowed at EL1.

If any of the following are true, then this field is ignored by the PE:

- SelfHostedTraceEnabled() == FALSE.
- The Effective value of HCR_EL2.TGE is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

E0TRE, bit [0]

EL0 Trace Enable.

E0TRE	Meaning
0b0	Trace is prohibited at EL0.
0b1	Trace is allowed at EL0.

If any of the following are true, then this field is ignored by the PE:

- SelfHostedTraceEnabled() == FALSE.
- The Effective value of HCR_EL2.TGE is 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Accessing TRFCR_EL1

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, TRFCR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0010	0b001

```
if !IsFeatureImplemented(FEAT_TRF) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.TTRF == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = TRFCR_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = TRFCR_EL1;
```

MSR TRFCR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0010	0b001

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_TRF) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.TTRF == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        TRFCR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    TRFCR_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.4.2 TRFCR_EL2, Trace Filter Control Register (EL2)

The TRFCR_EL2 characteristics are:

Purpose

Provides EL2 controls for Trace.

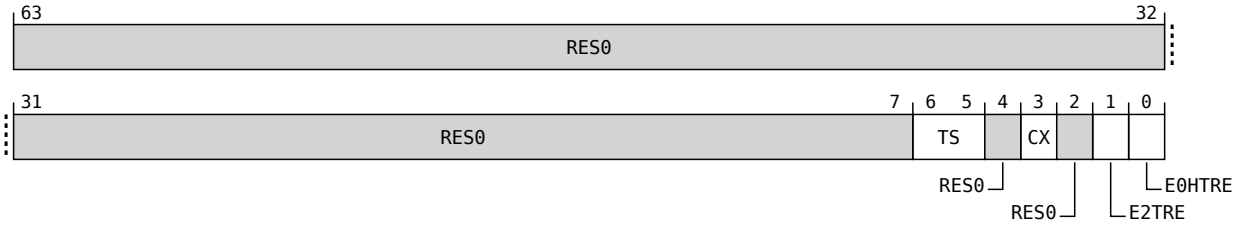
Configuration

This register is present only when FEAT_TRF is implemented. Otherwise, direct accesses to TRFCR_EL2 are UNDEFINED.

Attributes

TRFCR_EL2 is a 64-bit register.

Field descriptions



Bits [63:7]

Reserved, RES0.

TS, bits [6:5]

Timestamp Control. Controls which timebase is used for trace timestamps.

TS	Meaning
0b00	Timestamp controlled by TRFCR_EL1.TS.
0b01	Virtual timestamp. The traced timestamp is the physical counter value minus the value of CNTVOFF_EL2.
0b11	Physical timestamp. The traced timestamp is the physical counter value.

If SelfHostedTraceEnabled() == FALSE, then this field is ignored by the PE.

The reset behavior of this field is:

- On a Warm reset, this field resets to '00'.

Bit [4]

Reserved, RES0.

CX, bit [3]

CONTEXTIDR_EL2 and VMID trace enable.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CX	Meaning
0b0	CONTEXTIDR_EL2 and VMID trace prohibited.
0b1	CONTEXTIDR_EL2 and VMID trace allowed.

If SelfHostedTraceEnabled() == FALSE, then this field is ignored by the PE.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Bit [2]

Reserved, RES0.

E2TRE, bit [1]

EL2 Trace Enable.

E2TRE	Meaning
0b0	Trace is prohibited at EL2.
0b1	Trace is allowed at EL2.

If SelfHostedTraceEnabled() == FALSE, then this field is ignored by the PE.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

E0HTRE, bit [0]

EL0 Trace Enable.

E0HTRE	Meaning
0b0	Trace is prohibited at EL0.
0b1	Trace is allowed at EL0.

If any of the following are true, then this field is ignored by the PE:

- SelfHostedTraceEnabled() == FALSE.
- The Effective value of HCR_EL2.TGE is 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Accessing TRFCR_EL2

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, TRFCR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0010	0b001

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_TRF) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = TRFCR_EL2;
```

MSR TRFCR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0010	0b001

```
if !IsFeatureImplemented(FEAT_TRF) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    TRFCR_EL2 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.5 Performance Monitors registers

This section lists the Performance Monitoring registers in AArch64.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.5.1 PMCCFILTR_EL0, Performance Monitors Cycle Count Filter Register

The PMCCFILTR_EL0 characteristics are:

Purpose

Determines the modes in which the Cycle Counter, PMCCNTR_EL0, increments.

Configuration

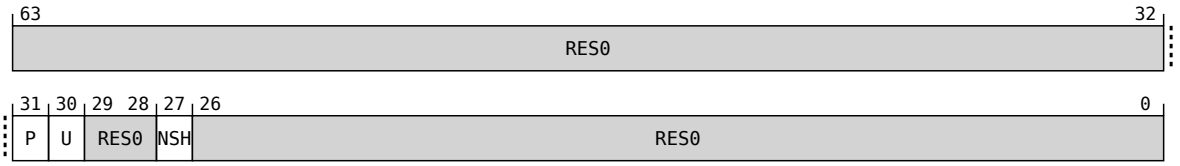
AArch64 System register PMCCFILTR_EL0 bits [31:0] are architecturally mapped to External register PMCCFILTR_EL0[31:0].

This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCCFILTR_EL0 are UNDEFINED.

Attributes

PMCCFILTR_EL0 is a 64-bit register.

Field descriptions



Bits [63:32]

Reserved, RES0.

P, bit [31]

EL1 filtering. Controls counting cycles in EL1.

P	Meaning
0b0	This mechanism has no effect on filtering of cycles.
0b1	The PE does not count cycles in EL1.

If Secure and Non-secure states are implemented, then counting cycles in Non-secure EL1 is further controlled by PMCCFILTR_EL0.NSK.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

U, bit [30]

EL0 filtering. Controls counting cycles in EL0.

U	Meaning
0b0	This mechanism has no effect on filtering of cycles.
0b1	The PE does not count cycles in EL0.

If Secure and Non-secure states are implemented, then counting cycles in Non-secure EL0 is further controlled by PMCCFILTR_EL0.NSU.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [29:28]

Reserved, RES0.

NSH, bit [27]

EL2 filtering. Controls counting cycles in EL2.

NSH	Meaning
0b0	The PE does not count cycles in EL2.
0b1	This mechanism has no effect on filtering of cycles.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [26:0]

Reserved, RES0.

Accessing PMCCFILTR_EL0

PMCCFILTR_EL0 can also be accessed by using PMXEVTYPER_EL0 with PMSELR_EL0.SEL set to 0b11111.

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, PMCCFILTR_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b1111	0b111

```
if !IsFeatureImplemented(FEAT_PMUv3) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if PMUSERENR_EL0.EN == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = PMCCFILTR_EL0;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = PMCCFILTR_EL0;
elsif PSTATE.EL == EL2 then
    X[t, 64] = PMCCFILTR_EL0;
```

MSR PMCCFILTR_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b1111	0b111

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_PMUv3) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if PMUSERENR_EL0.EN == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        PMCCFILTR_EL0 = X[t, 64];
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        PMCCFILTR_EL0 = X[t, 64];
elsif PSTATE.EL == EL2 then
    PMCCFILTR_EL0 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.5.2 PMCCNTR_EL0, Performance Monitors Cycle Count Register

The PMCCNTR_EL0 characteristics are:

Purpose

Holds the value of the processor Cycle Counter, CCNT, that counts processor clock cycles. See 'Time as measured by the Performance Monitors cycle counter' for more information.

PMCCFILTR_EL0 determines the modes and states in which the PMCCNTR_EL0 can increment.

Configuration

All counters are subject to any changes in clock frequency, including clock stopping caused by the WFI and WFE instructions. This means that it is CONSTRAINED UNPREDICTABLE whether or not PMCCNTR_EL0 continues to increment when clocks are stopped by WFI and WFE instructions.

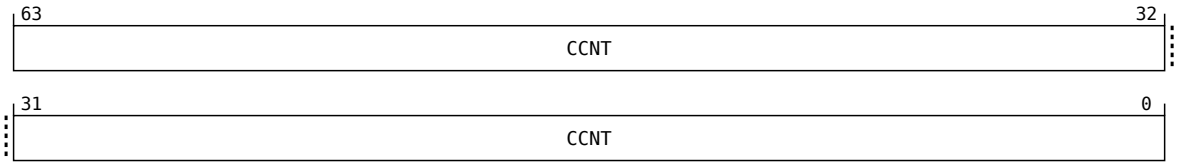
AArch64 System register PMCCNTR_EL0 bits [63:0] are architecturally mapped to External register PMCCNTR_EL0[63:0].

This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCCNTR_EL0 are UNDEFINED.

Attributes

PMCCNTR_EL0 is a 64-bit register.

Field descriptions



CCNT, bits [63:0]

Cycle count. Depending on the values of PMCR_EL0.{LC,D}, this field increments in one of the following ways:

- Every processor clock cycle.
- Every 64th processor clock cycle.

Writing 1 to PMCR_EL0.C sets this field to 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PMCCNTR_EL0

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, PMCCNTR_EL0`

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1101	0b000

```
if !IsFeatureImplemented(FEAT_PMUv3) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if PMUSERENR_EL0.<CR,EN> == '00' then
        if HCR_EL2.TGE == '1' then
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = PMCCNTR_EL0;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = PMCCNTR_EL0;
elsif PSTATE.EL == EL2 then
    X[t, 64] = PMCCNTR_EL0;
```

MSR PMCCNTR_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1101	0b000

```
if !IsFeatureImplemented(FEAT_PMUv3) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if PMUSERENR_EL0.EN == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        PMCCNTR_EL0 = X[t, 64];
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        PMCCNTR_EL0 = X[t, 64];
elsif PSTATE.EL == EL2 then
    PMCCNTR_EL0 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.5.3 PMCEID0_EL0, Performance Monitors Common Event Identification Register 0

The PMCEID0_EL0 characteristics are:

Purpose

Defines which Common architectural events and Common microarchitectural events are implemented, or counted, using PMU events in the ranges `0x0000` to `0x001F` and `0x4000` to `0x401F`.

For more information about the Common events and the use of the PMCEID<n>_EL0 registers see 'The PMU event number space and common events'.

Configuration

AArch64 System register PMCEID0_EL0 bits [31:0] are architecturally mapped to External register PMCEID0[31:0].

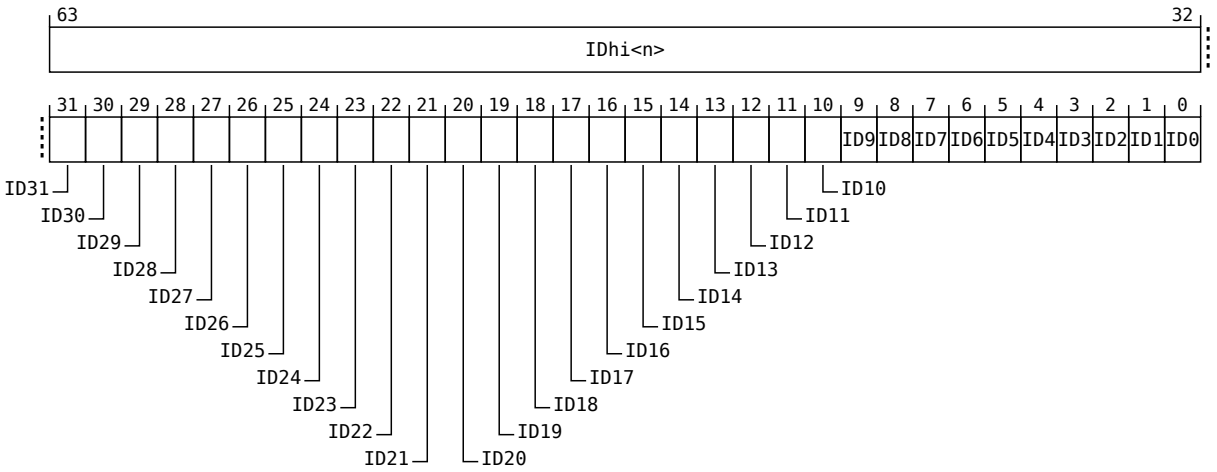
AArch64 System register PMCEID0_EL0 bits [63:32] are architecturally mapped to External register PMCEID2[31:0].

This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCEID0_EL0 are UNDEFINED.

Attributes

PMCEID0_EL0 is a 64-bit register.

Field descriptions



IDhi<n> , bits [n+32], for n = 31 to 0

When FEAT_PMUv3p1 is implemented:

IDhi[n] corresponds to Common event (`0x4000` + n).

For each bit:

IDhi<n>	Meaning
`0b0`	The Common event is not implemented, or not counted.
`0b1`	The Common event is implemented.

When the value of a bit in the field is 1, the corresponding Common event is implemented and counted.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Arm recommends that if a Common event is never counted, the value of the corresponding bit is 0.

A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional Common event.

Note

Such an event might be added retrospectively to an earlier version of the PMU architecture, provided the event does not require any additional PMU features and has an event number that can be represented in the PMCEID<n>_EL0 registers of that earlier version of the PMU architecture.

Otherwise:

Reserved, RES0.

ID<n> , bits [n], for n = 31 to 0

ID[n] corresponds to Common event n.

For each bit:

ID<n>	Meaning
0b0	The Common event is not implemented, or not counted.
0b1	The Common event is implemented.

When the value of a bit in the field is 1, the corresponding Common event is implemented and counted.

Note

Arm recommends that if a Common event is never counted, the value of the corresponding bit is 0.

A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional Common event.

Note

Such an event might be added retrospectively to an earlier version of the PMU architecture, provided the event does not require any additional PMU features and has an event number that can be represented in the PMCEID<n>_EL0 registers of that earlier version of the PMU architecture.

Accessing PMCEID0_EL0

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, PMCEID0_EL0`

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1100	0b110

```
if !IsFeatureImplemented(FEAT_PMUv3) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if PMUSERENR_EL0.EN == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        X[t, 64] = PMCEID0_EL0;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = PMCEID0_EL0;
elsif PSTATE.EL == EL2 then
    X[t, 64] = PMCEID0_EL0;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.5.4 PMCEID1_EL0, Performance Monitors Common Event Identification Register 1

The PMCEID1_EL0 characteristics are:

Purpose

Defines which Common architectural events and Common microarchitectural events are implemented, or counted, using PMU events in the ranges `0x0020` to `0x003F` and `0x4020` to `0x403F`.

For more information about the Common events and the use of the PMCEID<n>_EL0 registers see 'The PMU event number space and common events'.

Configuration

AArch64 System register PMCEID1_EL0 bits [31:0] are architecturally mapped to External register PMCEID1[31:0].

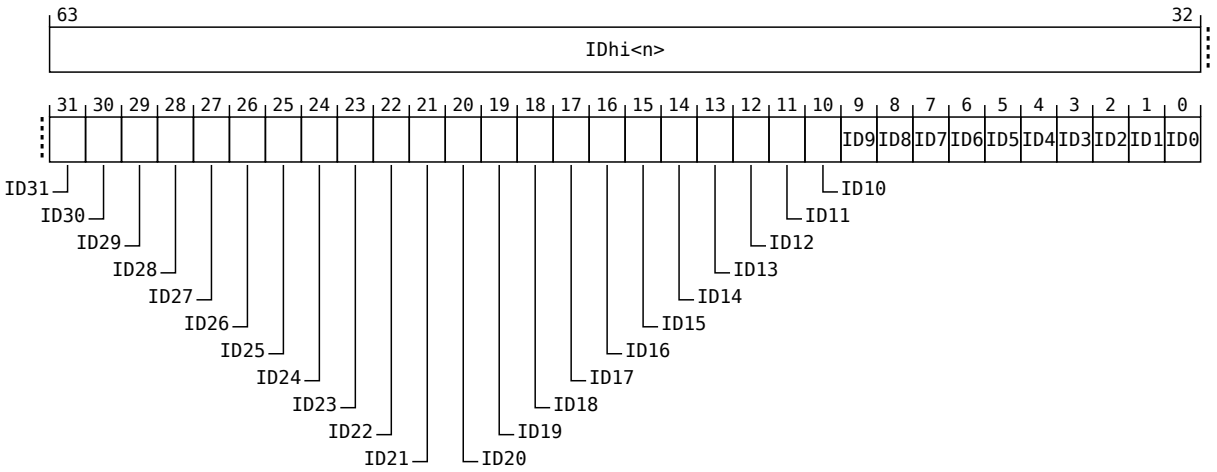
AArch64 System register PMCEID1_EL0 bits [63:32] are architecturally mapped to External register PMCEID3[31:0].

This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCEID1_EL0 are UNDEFINED.

Attributes

PMCEID1_EL0 is a 64-bit register.

Field descriptions



IDhi<n> , bits [n+32], for n = 31 to 0

When FEAT_PMUv3p1 is implemented:

IDhi[n] corresponds to Common event (`0x4020` + n).

For each bit:

IDhi<n>	Meaning
`0b0`	The Common event is not implemented, or not counted.
`0b1`	The Common event is implemented.

When the value of a bit in the field is 1, the corresponding Common event is implemented and counted.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Arm recommends that if a Common event is never counted, the value of the corresponding bit is 0.

A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional Common event.

Note

Such an event might be added retrospectively to an earlier version of the PMU architecture, provided the event does not require any additional PMU features and has an event number that can be represented in the PMCEID<n>_EL0 registers of that earlier version of the PMU architecture.

Otherwise:

Reserved, RES0.

ID<n> , bits [n], for n = 31 to 0

ID[n] corresponds to Common event (0x0020 + n).

For each bit:

ID<n>	Meaning
0b0	The Common event is not implemented, or not counted.
0b1	The Common event is implemented.

When the value of a bit in the field is 1, the corresponding Common event is implemented and counted.

Note

Arm recommends that if a Common event is never counted, the value of the corresponding bit is 0.

A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional Common event.

Note

Such an event might be added retrospectively to an earlier version of the PMU architecture, provided the event does not require any additional PMU features and has an event number that can be represented in the PMCEID<n>_EL0 registers of that earlier version of the PMU architecture.

Accessing PMCEID1_EL0

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PMCEID1_EL0
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1100	0b111

```
if !IsFeatureImplemented(FEAT_PMUv3) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if PMUSERENR_EL0.EN == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        X[t, 64] = PMCEID1_EL0;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = PMCEID1_EL0;
elsif PSTATE.EL == EL2 then
    X[t, 64] = PMCEID1_EL0;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.5.5 PMCNTENCLR_EL0, Performance Monitors Count Enable Clear Register

The PMCNTENCLR_EL0 characteristics are:

Purpose

Allows software to disable the following counters:

- The cycle counter PMCCNTR_EL0.
- The event counters PMEVCNTR<n>_EL0.

Reading from this register shows which counters are enabled.

Configuration

AArch64 System register PMCNTENCLR_EL0 bits [63:0] are architecturally mapped to AArch64 System register PMCNTENSET_EL0[63:0].

AArch64 System register PMCNTENCLR_EL0 bits [31:0] are architecturally mapped to External register PMCNTENCLR_EL0[31:0].

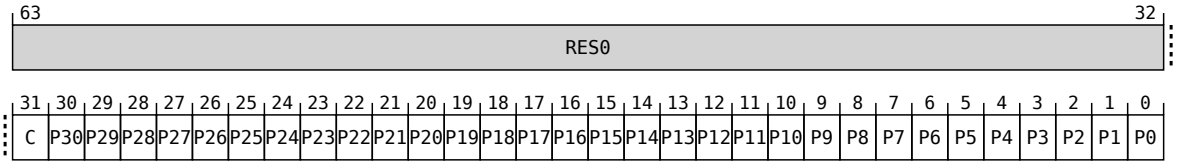
AArch64 System register PMCNTENCLR_EL0 bits [31:0] are architecturally mapped to External register PMCNTENSET_EL0[31:0].

This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCNTENCLR_EL0 are UNDEFINED.

Attributes

PMCNTENCLR_EL0 is a 64-bit register.

Field descriptions



Bits [63:32]

Reserved, RES0.

C, bit [31]

PMCCNTR_EL0 disable. On writes, allows software to disable PMCCNTR_EL0. On reads, returns the PMCCNTR_EL0 enable status.

C	Meaning
0b0	PMCCNTR_EL0 disabled.
0b1	PMCCNTR_EL0 enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is W1C.

P<m> , bits [m], for m = 30 to 0

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PMEVCNTR<m>_EL0 disable. On writes, allows software to disable PMEVCNTR<m>_EL0. On reads, returns the PMEVCNTR<m>_EL0 enable status.

P<m>	Meaning
0b0	PMEVCNTR<m>_EL0 disabled.
0b1	PMEVCNTR<m>_EL0 enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= GetNumEventCountersAccessible(), access to this field is RAZ/WI.
- Otherwise, access to this field is W1C.

Accessing PMCNTENCLR_EL0

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, PMCNTENCLR_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1100	0b010

```
if !IsFeatureImplemented(FEAT_PMUv3) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if PMUSERENR_EL0.EN == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = PMCNTENCLR_EL0;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = PMCNTENCLR_EL0;
elsif PSTATE.EL == EL2 then
    X[t, 64] = PMCNTENCLR_EL0;
```

MSR PMCNTENCLR_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1100	0b010

```
if !IsFeatureImplemented(FEAT_PMUv3) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if PMUSERENR_EL0.EN == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        PMCNTENCLR_EL0 = X[t, 64];
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        PMCNTENCLR_EL0 = X[t, 64];
elsif PSTATE.EL == EL2 then
    PMCNTENCLR_EL0 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.5.6 PMCNTENSET_EL0, Performance Monitors Count Enable Set Register

The PMCNTENSET_EL0 characteristics are:

Purpose

Allows software to enable the following counters:

- The cycle counter PMCCNTR_EL0.
- The event counters PMEVCNTR<n>_EL0.

Reading from this register shows which counters are enabled.

Configuration

AArch64 System register PMCNTENSET_EL0 bits [63:0] are architecturally mapped to AArch64 System register PMCNTENCLR_EL0[63:0].

AArch64 System register PMCNTENSET_EL0 bits [31:0] are architecturally mapped to External register PMCNTENSET_EL0[31:0].

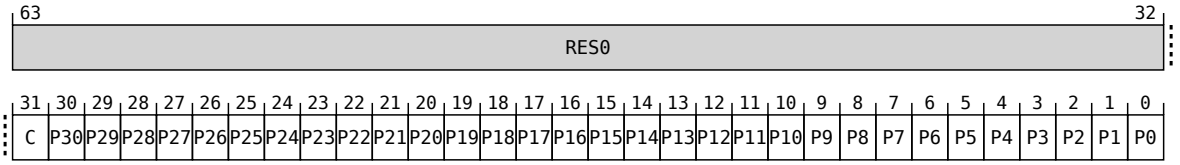
AArch64 System register PMCNTENSET_EL0 bits [31:0] are architecturally mapped to External register PMCNTENCLR_EL0[31:0].

This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCNTENSET_EL0 are UNDEFINED.

Attributes

PMCNTENSET_EL0 is a 64-bit register.

Field descriptions



Bits [63:32]

Reserved, RES0.

C, bit [31]

PMCCNTR_EL0 enable. On writes, allows software to enable PMCCNTR_EL0. On reads, returns the PMCCNTR_EL0 enable status.

C	Meaning
0b0	PMCCNTR_EL0 disabled.
0b1	PMCCNTR_EL0 enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is W1S.

P<m> , bits [m], for m = 30 to 0

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PMEVCNTR<m>_EL0 enable. On writes, allows software to enable PMEVCNTR<m>_EL0. On reads, returns the PMEVCNTR<m>_EL0 enable status.

P<m>	Meaning
0b0	PMEVCNTR<m>_EL0 disabled.
0b1	PMEVCNTR<m>_EL0 enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= GetNumEventCountersAccessible(), access to this field is RAZ/WI.
- Otherwise, access to this field is W1S.

Accessing PMCNTENSET_EL0

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, PMCNTENSET_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1100	0b001

```
if !IsFeatureImplemented(FEAT_PMUv3) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if PMUSERENR_EL0.EN == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = PMCNTENSET_EL0;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = PMCNTENSET_EL0;
elsif PSTATE.EL == EL2 then
    X[t, 64] = PMCNTENSET_EL0;
```

MSR PMCNTENSET_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1100	0b001

```
if !IsFeatureImplemented(FEAT_PMUv3) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if PMUSERENR_EL0.EN == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        PMCNTENSET_EL0 = X[t, 64];
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        PMCNTENSET_EL0 = X[t, 64];
elsif PSTATE.EL == EL2 then
    PMCNTENSET_EL0 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.5.7 PMCR_EL0, Performance Monitors Control Register

The PMCR_EL0 characteristics are:

Purpose

Provides details of the Performance Monitors implementation, including the number of counters implemented, and configures and controls the counters.

Configuration

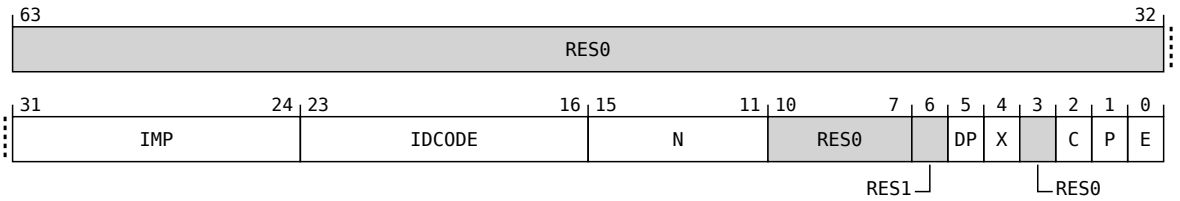
AArch64 System register PMCR_EL0 bits [31:0] are architecturally mapped to External register PMCR_EL0[31:0].

This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMCR_EL0 are UNDEFINED.

Attributes

PMCR_EL0 is a 64-bit register.

Field descriptions



Bits [63:32]

Reserved, RES0.

IMP, bits [31:24]

Implementer code.

If this field is zero, then PMCR_EL0.IDCODE is RES0 and software must use MIDR_EL1 to identify the PE.

Otherwise, this field and PMCR_EL0.IDCODE identify the PMU implementation to software. The implementer codes are allocated by Arm. A nonzero value has the same interpretation as MIDR_EL1.Implementer.

This field has an IMPLEMENTATION DEFINED value.

Arm deprecates use of this field.

Access to this field is RO.

IDCODE, bits [23:16]

When PMCR_EL0.IMP != '00000000':

Identification code. Arm deprecates use of this field.

Each implementer must maintain a list of identification codes that are specific to the implementer. A specific implementation is identified by the combination of the implementer code and the identification code.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Otherwise:

Reserved, RES0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

N, bits [15:11]

Indicates the number of event counters implemented. This value is in the range of `0b00000`-`0b11111`. If the value is `0b00000`, then only PMCCNTR_EL0 is implemented. If the value is `0b11111`, then PMCCNTR_EL0 and 31 event counters are implemented.

When EL2 is implemented and enabled for the current Security state, reads of this field from EL1 and EL0 return the Effective value of MDCR_EL2.HPMN.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Bits [10:7]

Reserved, RES0.

Bit [6]

Reserved, RES1.

DP, bit [5]

When FEAT_PMUv3p1 is implemented:

Disable cycle counter when event counting is prohibited.

DP	Meaning
`0b0`	Cycle counting by PMCCNTR_EL0 is not affected by this mechanism.
`0b1`	Cycle counting by PMCCNTR_EL0 is disabled in prohibited regions: • If FEAT_PMUv3p1 is implemented, EL2 is implemented, and MDCR_EL2.HPMD is 1, then cycle counting by PMCCNTR_EL0 is disabled at EL2.

The conditions when this field disables the cycle counter are the same as when event counting by an event counter in the first range is prohibited. For more information about event counter ranges, see MDCR_EL2.HPMN.

For more information, see 'Prohibiting event and cycle counting'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

X, bit [4]

When the implementation includes a PMU event export bus:

Enable export of events in an IMPLEMENTATION DEFINED PMU event export bus.

X	Meaning
`0b0`	Do not export events.
`0b1`	Export events where not prohibited.

This field enables the exporting of events over an IMPLEMENTATION DEFINED PMU event export bus to another device.

No events are exported when counting is prohibited.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This field does not affect the generation of Performance Monitors overflow interrupt requests or signaling to a cross-trigger interface (CTI) that can be implemented as signals exported from the PE.

If FEAT_ETMv4 is implemented, this field does affect the use of PMU events as an External Input by the trace unit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

Bit [3]

Reserved, RES0.

C, bit [2]

Cycle counter reset. The effects of writing to this field are:

C	Meaning
0b0	No action.
0b1	Reset PMCCNTR_EL0 to zero.

Note

Resetting PMCCNTR_EL0 does not change the cycle counter overflow field. The value of PMCR_EL0.LC is ignored, and bits [63:0] of the cycle counter are reset.

Access to this field is WO/RAZ.

P, bit [1]

Event counter reset.

P	Meaning
0b0	No action.
0b1	Reset all affected event counters PMEVCNTR<n>_EL0 to zero.

The event counters affected by this field are:

- All event counters in the first range.
- If any of the following are true, all event counters in the second range:
  - EL2 is not implemented.
  - The PE is executing at EL2.

Writes to this field do not affect other event counters or the cycle counter PMCCNTR_EL0.

For more information about event counter ranges, see MDCR_EL2.HPMN.

Note

Resetting the event counters does not change the event counter overflow fields.

Access to this field is WO/RAZ.

E, bit [0]

Enable.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E	Meaning
0b0	Affected counters are disabled and do not count.
0b1	Affected counters are enabled by PMCNTENSET_EL0.

The counters affected by this field are:

- The event counters in the first range. For more information about event counter ranges, see MDCR_EL2.HPMN.
- The cycle counter PMCCNTR_EL0.

Other event counters are not affected by this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Accessing PMCR_EL0

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, PMCR_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1100	0b000

```
if !IsFeatureImplemented(FEAT_PMUv3) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if PMUSERENR_EL0.EN == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif MDCR_EL2.TPMCR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = PMCR_EL0;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif MDCR_EL2.TPMCR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = PMCR_EL0;
elsif PSTATE.EL == EL2 then
    X[t, 64] = PMCR_EL0;
```

MSR PMCR_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1100	0b000

```
if !IsFeatureImplemented(FEAT_PMUv3) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if PMUSERENR_EL0.EN == '0' then
        if HCR_EL2.TGE == '1' then
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif MDCR_EL2.TPMCR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        PMCR_EL0 = X[t, 64];
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif MDCR_EL2.TPMCR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        PMCR_EL0 = X[t, 64];
elsif PSTATE.EL == EL2 then
    PMCR_EL0 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.5.8 PMEVCNTR<n>_EL0, Performance Monitors Event Count Registers, n = 0 - 30

The PMEVCNTR<n>_EL0 characteristics are:

Purpose

Holds event counter n, which counts events, where n is 0 to 30.

Configuration

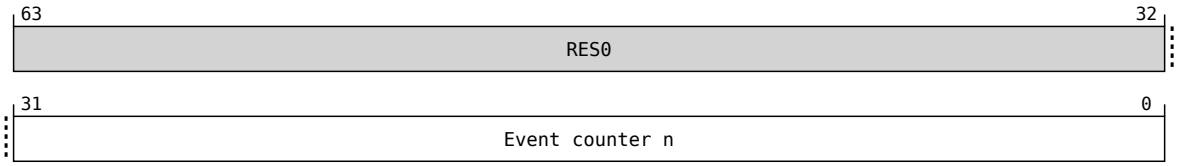
AArch64 System register PMEVCNTR<n>_EL0 bits [31:0] are architecturally mapped to External register PMEVCNTR_EL0[31:0].

This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMEVCNTR<n>_EL0 are UNDEFINED.

Attributes

PMEVCNTR<n>_EL0 is a 64-bit register.

Field descriptions



Bits [63:32]

Reserved, RES0.

EVCNT, bits [31:0]

Event counter n. Value of event counter n, where n is the number of this register and is a number from 0 to 30.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PMEVCNTR<n>_EL0

PMEVCNTR<n>_EL0 can also be accessed by using PMXEVCNTR_EL0 with PMSELR_EL0.SEL set to the value of <n>.

If <n> is greater than or equal to the number of accessible event counters, then reads and writes of PMEVCNTR<n>_EL0 are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP.
- Accesses to the register behave as if <n> is an UNKNOWN value less-than-or-equal-to the index of the highest accessible event counter.
- If EL2 is implemented and enabled in the current Security state, and <n> is less than the number of implemented event counters, accesses from EL1 or permitted accesses from EL0 are trapped to EL2.

Note

In EL0, an access is permitted if it is enabled by PMUSERENR_EL0.{ER,EN}.
In EL1 and EL0, MDCR_EL2.HPMN identifies the number of accessible event counters. Otherwise, the number of accessible event counters is the number of implemented event counters. For more information, see MDCR_EL2.HPMN.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MRS <Xt>, PMEVCNTR<m>_EL0 ; Where m = 0-30

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b10:m[4:3]	m[2:0]

```
integer m = UInt(CRm<1:0>:op2<2:0>);

if !IsFeatureImplemented(FEAT_PMUv3) then
    UNDEFINED;
elsif m >= GetNumEventCountersSelfHosted() then
    ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
elsif PSTATE.EL == EL0 then
    if PMUSERENR_EL0.<ER,EN> == '00' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif m >= GetNumEventCountersAccessible() then
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    else
        X[t, 64] = PMEVCNTR_EL0[m];
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif m >= GetNumEventCountersAccessible() then
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    else
        X[t, 64] = PMEVCNTR_EL0[m];
elsif PSTATE.EL == EL2 then
    X[t, 64] = PMEVCNTR_EL0[m];
```

MSR PMEVCNTR<m>_EL0, <Xt> ; Where m = 0-30

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b10:m[4:3]	m[2:0]

```
integer m = UInt(CRm<1:0>:op2<2:0>);

if !IsFeatureImplemented(FEAT_PMUv3) then
    UNDEFINED;
elsif m >= GetNumEventCountersSelfHosted() then
    ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
elsif PSTATE.EL == EL0 then
    if PMUSERENR_EL0.EN == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif m >= GetNumEventCountersAccessible() then
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    else
        PMEVCNTR_EL0[m] = X[t, 64];
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif m >= GetNumEventCountersAccessible() then
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    else
        PMEVCNTR_EL0[m] = X[t, 64];
elsif PSTATE.EL == EL2 then
    PMEVCNTR_EL0[m] = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.5.9 PMEVTYPER<n>_EL0, Performance Monitors Event Type Registers, n = 0 - 30

The PMEVTYPER<n>_EL0 characteristics are:

Purpose

Configures event counter n, where n is 0 to 30.

Configuration

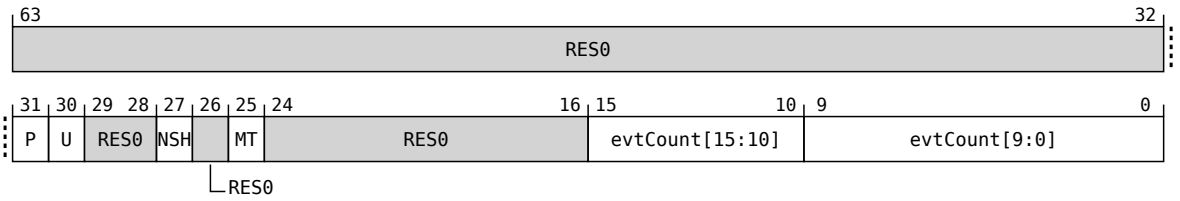
AArch64 System register PMEVTYPER<n>_EL0 bits [31:0] are architecturally mapped to External register PMEVTYPER_EL0[31:0].

This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMEVTYPER<n>_EL0 are UNDEFINED.

Attributes

PMEVTYPER<n>_EL0 is a 64-bit register.

Field descriptions



Bits [63:32]

Reserved, RES0.

P, bit [31]

EL1 filtering. Controls counting events in EL1.

P	Meaning
0b0	This mechanism has no effect on filtering of events.
0b1	The PE does not count events in EL1.

If Secure and Non-secure states are implemented, then counting events in Non-secure EL1 is further controlled by PMEVTYPER<n>_EL0.NSK.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

U, bit [30]

EL0 filtering. Controls counting events in EL0.

U	Meaning
0b0	This mechanism has no effect on filtering of events.
0b1	The PE does not count events in EL0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If Secure and Non-secure states are implemented, then counting events in Non-secure EL0 is further controlled by PMEVTYPER<n>_EL0.NSU.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [29:28]

Reserved, RES0.

NSH, bit [27]

EL2 filtering. Controls counting events in EL2.

NSH	Meaning
0b0	The PE does not count events in EL2.
0b1	This mechanism has no effect on filtering of events.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [26]

Reserved, RES0.

MT, bit [25]

When an IMPLEMENTATION DEFINED multi-threaded PMU extension is implemented:

Multithreading.

MT	Meaning
0b0	Count events only on controlling PE.
0b1	Count events from any PE with the same affinity at level 1 and above as this PE.

Unless otherwise stated:

- If the event counts PE cycles when a stall condition is true and a second condition is true, then the counter counts Processor cycles when the stall condition is true for all of these PEs, and the second condition is true for any of these PEs.
- If the event counts PE cycles when any other condition is true, then the counter counts Processor cycles when the condition is true for any of these PEs.
- Otherwise, the event counts by the sum of the count across all of these PEs.

For the stall events, the stall condition means the applicable condition described by the STALL, STALL_FRONTEND, or STALL_BACKEND event.

The second condition is any condition in addition to this.

For example, for the STALL_FRONTEND_L1I event, the stall condition is STALL_FRONTEND, and the second condition is when there is a demand instruction miss in the first level of instruction cache.

For the STALL, STALL_FRONTEND, and STALL_BACKEND events themselves, the second condition is the null TRUE condition.

See ‘Multithreaded implementations’ and ‘Cycle event counting in multithreaded implementations’.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bits [24:16]

Reserved, RES0.

evtCount[15:10], bits [15:10]

When FEAT_PMUv3p1 is implemented:

Extension to evtCount[9:0]. For more information, see evtCount[9:0].

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

evtCount[9:0], bits [9:0]

Event to count.

The event number of the event that is counted by event counter PMEVCNTR<n>_EL0.

The ranges of event numbers allocated to each type of event are shown in 'Allocation of the PMU event number space'.

If PMEVTYPER<n>_EL0.evtCount is programmed to an event that is reserved or not supported by the PE, the behavior depends on the value written:

- For the range 0x0000 to 0x003F, no events are counted and the value returned by a direct or external read of the PMEVTYPER<n>_EL0.evtCount field is the value written to the field.
- If FEAT_PMUv3p1 is implemented, for the range 0x4000 to 0x403F, no events are counted and the value returned by a direct or external read of the PMEVTYPER<n>_EL0.evtCount field is the value written to the field.
- For other values, it is UNPREDICTABLE what event, if any, is counted and the value returned by a direct or external read of the PMEVTYPER<n>_EL0.evtCount field is UNKNOWN.

Note

UNPREDICTABLE means the event must not expose privileged information.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PMEVTYPER<n>_EL0

PMEVTYPER<n>_EL0 can also be accessed by using PMXEVTYPER_EL0 with PMSELR_EL0.SEL set to n.

If <n> is greater than or equal to the number of accessible event counters, then reads and writes of PMEVTYPER<n>_EL0 are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP.
- Accesses to the register behave as if <n> is an UNKNOWN value less-than-or-equal-to the index of the highest accessible event counter.
- If EL2 is implemented and enabled in the current Security state, and <n> is less than the number of implemented event counters, accesses from EL1 or permitted accesses from EL0 are trapped to EL2.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

In EL0, an access is permitted if it is enabled by PMUSERENR_EL0.EN.
If EL2 is implemented and enabled in the current Security state, in EL1 and EL0, MDCR_EL2.HPMN identifies the number of accessible event counters. Otherwise, the number of accessible event counters is the number of implemented event counters. For more information, see MDCR_EL2.HPMN.

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, PMEVTYPER<m>_EL0 ; Where m = 0-30`

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b11:m[4:3]	m[2:0]

```
integer m = UInt(CRm<1:0>:op2<2:0>);

if !IsFeatureImplemented(FEAT_PMUv3) then
    UNDEFINED;
elsif m >= GetNumEventCountersSelfHosted() then
    ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
elsif PSTATE.EL == EL0 then
    if PMUSERENR_EL0.EN == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif m >= GetNumEventCountersAccessible() then
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    else
        X[t, 64] = PMEVTYPER_EL0[m];
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif m >= GetNumEventCountersAccessible() then
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    else
        X[t, 64] = PMEVTYPER_EL0[m];
elsif PSTATE.EL == EL2 then
    X[t, 64] = PMEVTYPER_EL0[m];
```

`MSR PMEVTYPER<m>_EL0, <Xt> ; Where m = 0-30`

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b11:m[4:3]	m[2:0]

```
integer m = UInt(CRm<1:0>:op2<2:0>);

if !IsFeatureImplemented(FEAT_PMUv3) then
    UNDEFINED;
elsif m >= GetNumEventCountersSelfHosted() then
    ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
elsif PSTATE.EL == EL0 then
    if PMUSERENR_EL0.EN == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif MDCR_EL2.TPM == '1' then
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif m >= GetNumEventCountersAccessible() then
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    else
        PMEVTYPER_EL0[m] = X[t, 64];
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif m >= GetNumEventCountersAccessible() then
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    else
        PMEVTYPER_EL0[m] = X[t, 64];
elsif PSTATE.EL == EL2 then
    PMEVTYPER_EL0[m] = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.5.10 PMINTENCLR_EL1, Performance Monitors Interrupt Enable Clear Register

The PMINTENCLR_EL1 characteristics are:

Purpose

Allows software to disable the generation of interrupt requests on overflows from the following counters:

- The cycle counter PMCCNTR_EL0.
- The event counters PMEVCNTR<n>_EL0.

Reading from this register shows which overflow interrupt requests are enabled.

Configuration

AArch64 System register PMINTENCLR_EL1 bits [63:0] are architecturally mapped to AArch64 System register PMINTENSET_EL1[63:0].

AArch64 System register PMINTENCLR_EL1 bits [31:0] are architecturally mapped to External register PMINTENCLR_EL1[31:0].

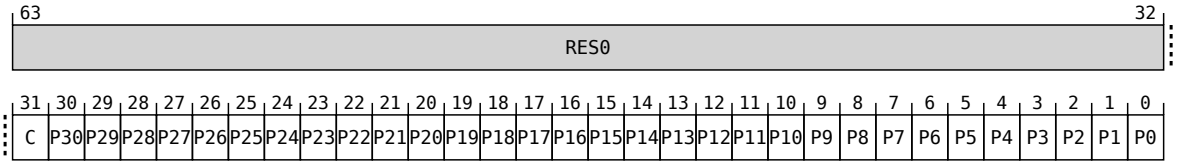
AArch64 System register PMINTENCLR_EL1 bits [31:0] are architecturally mapped to External register PMINTENSET_EL1[31:0].

This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMINTENCLR_EL1 are UNDEFINED.

Attributes

PMINTENCLR_EL1 is a 64-bit register.

Field descriptions



Bits [63:32]

Reserved, RES0.

C, bit [31]

Interrupt request on unsigned overflow of PMCCNTR_EL0 disable. On writes, allows software to disable the interrupt request on unsigned overflow of PMCCNTR_EL0. On reads, returns the interrupt request on unsigned overflow of PMCCNTR_EL0 enable status.

C	Meaning
0b0	Interrupt request on unsigned overflow of PMCCNTR_EL0 disabled.
0b1	Interrupt request on unsigned overflow of PMCCNTR_EL0 enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is W1C.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

P<m> , bits [m], for m = 30 to 0

Interrupt request on unsigned overflow of PMEVCNTR<m>_EL0 disable. On writes, allows software to disable the interrupt request on unsigned overflow of PMEVCNTR<m>_EL0. On reads, returns the interrupt request on unsigned overflow of PMEVCNTR<m>_EL0 enable status.

P<m>	Meaning
0b0	Interrupt request on unsigned overflow of PMEVCNTR<m>_EL0 disabled.
0b1	Interrupt request on unsigned overflow of PMEVCNTR<m>_EL0 enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= GetNumEventCountersAccessible(), access to this field is RAZ/WI.
- Otherwise, access to this field is W1C.

Accessing PMINTENCLR_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, PMINTENCLR_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1110	0b010

```
if !IsFeatureImplemented(FEAT_PMUv3) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = PMINTENCLR_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = PMINTENCLR_EL1;
```

`MSR PMINTENCLR_EL1, <Xt>`

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1110	0b010

```
if !IsFeatureImplemented(FEAT_PMUv3) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        PMINTENCLR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    PMINTENCLR_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.5.11 PMINTENSET_EL1, Performance Monitors Interrupt Enable Set Register

The PMINTENSET_EL1 characteristics are:

Purpose

Allows software to enable the generation of interrupt requests on overflows from the following counters:

- The cycle counter PMCCNTR_EL0.
- The event counters PMEVCNTR<n>_EL0.

Reading from this register shows which overflow interrupt requests are enabled.

Configuration

AArch64 System register PMINTENSET_EL1 bits [63:0] are architecturally mapped to AArch64 System register PMINTENCLR_EL1[63:0].

AArch64 System register PMINTENSET_EL1 bits [31:0] are architecturally mapped to External register PMINTENSET_EL1[31:0].

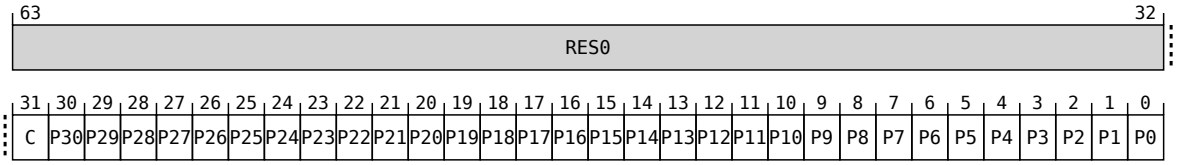
AArch64 System register PMINTENSET_EL1 bits [31:0] are architecturally mapped to External register PMINTENCLR_EL1[31:0].

This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMINTENSET_EL1 are UNDEFINED.

Attributes

PMINTENSET_EL1 is a 64-bit register.

Field descriptions



Bits [63:32]

Reserved, RES0.

C, bit [31]

Interrupt request on unsigned overflow of PMCCNTR_EL0 enable. On writes, allows software to enable the interrupt request on unsigned overflow of PMCCNTR_EL0. On reads, returns the interrupt request on unsigned overflow of PMCCNTR_EL0 enable status.

C	Meaning
0b0	Interrupt request on unsigned overflow of PMCCNTR_EL0 disabled.
0b1	Interrupt request on unsigned overflow of PMCCNTR_EL0 enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is W1S.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

P<m> , bits [m], for m = 30 to 0

Interrupt request on unsigned overflow of PMEVCNTR<m>_EL0 enable. On writes, allows software to enable the interrupt request on unsigned overflow of PMEVCNTR<m>_EL0. On reads, returns the interrupt request on unsigned overflow of PMEVCNTR<m>_EL0 enable status.

P<m>	Meaning
0b0	Interrupt request on unsigned overflow of PMEVCNTR<m>_EL0 disabled.
0b1	Interrupt request on unsigned overflow of PMEVCNTR<m>_EL0 enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= GetNumEventCountersAccessible(), access to this field is RAZ/WI.
- Otherwise, access to this field is W1S.

Accessing PMINTENSET_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, PMINTENSET_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1110	0b001

```
if !IsFeatureImplemented(FEAT_PMUv3) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = PMINTENSET_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = PMINTENSET_EL1;
```

`MSR PMINTENSET_EL1, <Xt>`

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1110	0b001

```
if !IsFeatureImplemented(FEAT_PMUv3) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        PMINTENSET_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    PMINTENSET_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.5.12 PMMIR_EL1, Performance Monitors Machine Identification Register

The PMMIR_EL1 characteristics are:

Purpose

Describes Performance Monitors parameters specific to the implementation to software.

Configuration

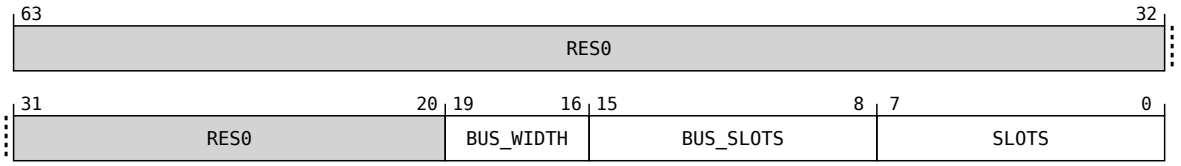
When FEAT_PMUv3_EXT is implemented and FEAT_PMUv3p4 is implemented, AArch64 System register PMMIR_EL1 bits [31:0] are architecturally mapped to External register PMMIR[31:0].

This register is present only when FEAT_PMUv3p4 is implemented. Otherwise, direct accesses to PMMIR_EL1 are UNDEFINED.

Attributes

PMMIR_EL1 is a 64-bit register.

Field descriptions



Bits [63:20]

Reserved, RES0.

BUS_WIDTH, bits [19:16]

Bus width. Indicates the number of bytes each BUS_ACCESS event relates to. Encoded as Log_2(number of bytes), plus one.

The value of this field is an IMPLEMENTATION DEFINED choice of:

BUS_WIDTH	Meaning
0b0000	The information is not available.
0b0011	Four bytes.
0b0100	8 bytes.
0b0101	16 bytes.
0b0110	32 bytes.
0b0111	64 bytes.
0b1000	128 bytes.
0b1001	256 bytes.
0b1010	512 bytes.
0b1011	1024 bytes.
0b1100	2048 bytes.

All other values are reserved.

Each transfer is up to this number of bytes. An access might be smaller than the bus width.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When this field is nonzero, each access counted by BUS_ACCESS is at most BUS_WIDTH bytes. An implementation might treat a wide bus as multiple narrower buses, such that a wide access on the bus increments the BUS_ACCESS counter by more than one.

Access to this field is RO.

BUS_SLOTS, bits [15:8]

Bus count. The largest value by which the BUS_ACCESS event might increment in a single BUS_CYCLES cycle.

This field has an IMPLEMENTATION DEFINED value.

When this field is nonzero, the largest value by which the BUS_ACCESS event might increment in a single BUS_CYCLES cycle is BUS_SLOTS.

If the bus count information is not available, this field will read as zero.

Access to this field is RO.

SLOTS, bits [7:0]

Operation width. The largest value by which the STALL_SLOT event might increment in a single cycle. If the STALL_SLOT event is not implemented, this field might read as zero.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing PMMIR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PMMIR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1110	0b110

```
if !IsFeatureImplemented(FEAT_PMUv3p4) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = PMMIR_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = PMMIR_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.5.13 PMOVSCLR_EL0, Performance Monitors Overflow Flag Status Clear Register

The PMOVSCLR_EL0 characteristics are:

Purpose

Allows software to clear the unsigned overflow flags for the following counters to 0:

- The cycle counter PMCCNTR_EL0.
- The event counters PMEVCNTR<n>_EL0.

Reading from this register shows the current unsigned overflow flag values.

Configuration

AArch64 System register PMOVSCLR_EL0 bits [63:0] are architecturally mapped to AArch64 System register PMOVSSET_EL0[63:0].

AArch64 System register PMOVSCLR_EL0 bits [31:0] are architecturally mapped to External register PMOVSCLR_EL0[31:0].

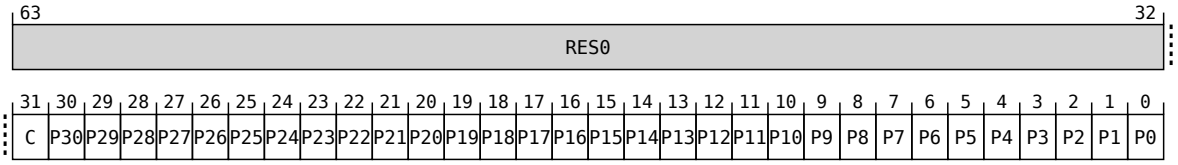
AArch64 System register PMOVSCLR_EL0 bits [31:0] are architecturally mapped to External register PMOVSSET_EL0[31:0].

This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMOVSCLR_EL0 are UNDEFINED.

Attributes

PMOVSCLR_EL0 is a 64-bit register.

Field descriptions



Bits [63:32]

Reserved, RES0.

C, bit [31]

Unsigned overflow flag for PMCCNTR_EL0 clear. On writes, allows software to clear the unsigned overflow flag for PMCCNTR_EL0 to 0. On reads, returns the unsigned overflow flag for PMCCNTR_EL0 overflow status.

C	Meaning
0b0	PMCCNTR_EL0 has not overflowed.
0b1	PMCCNTR_EL0 has overflowed.

PMCR_EL0.LC controls whether an overflow is detected from unsigned overflow of PMCCNTR_EL0[31:0] or unsigned overflow of PMCCNTR_EL0[63:0].

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is W1C.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

P<m> , bits [m], for m = 30 to 0

Unsigned overflow flag for PMEVCNTR<m>_EL0 clear. On writes, allows software to clear the unsigned overflow flag for PMEVCNTR<m>_EL0 to 0. On reads, returns the unsigned overflow flag for PMEVCNTR<m>_EL0 overflow status.

P<m>	Meaning
0b0	PMEVCNTR<m>_EL0 has not overflowed.
0b1	PMEVCNTR<m>_EL0 has overflowed.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= GetNumEventCountersAccessible(), access to this field is RAZ/WI.
- Otherwise, access to this field is W1C.

Accessing PMOVSCLR_EL0

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, PMOVSCLR_EL0`

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1100	0b011

```
if !IsFeatureImplemented(FEAT_PMUv3) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if PMUSERENR_EL0.EN == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = PMOVSCLR_EL0;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = PMOVSCLR_EL0;
elsif PSTATE.EL == EL2 then
    X[t, 64] = PMOVSCLR_EL0;
```

`MSR PMOVSCLR_EL0, <Xt>`

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1100	0b011

```
if !IsFeatureImplemented(FEAT_PMUv3) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if PMUSERENR_EL0.EN == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        PMOVSCLR_EL0 = X[t, 64];
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        PMOVSCLR_EL0 = X[t, 64];
elsif PSTATE.EL == EL2 then
    PMOVSCLR_EL0 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.5.14 PMOVSSET_EL0, Performance Monitors Overflow Flag Status Set Register

The PMOVSSET_EL0 characteristics are:

Purpose

Allows software to set the unsigned overflow flags for the following counters to 1:

- The cycle counter PMCCNTR_EL0.
- The event counters PMEVCNTR<n>_EL0.

Reading from this register shows the current unsigned overflow flag values.

Configuration

AArch64 System register PMOVSSET_EL0 bits [63:0] are architecturally mapped to AArch64 System register PMOVSCLR_EL0[63:0].

AArch64 System register PMOVSSET_EL0 bits [31:0] are architecturally mapped to External register PMOVSCLR_EL0[31:0].

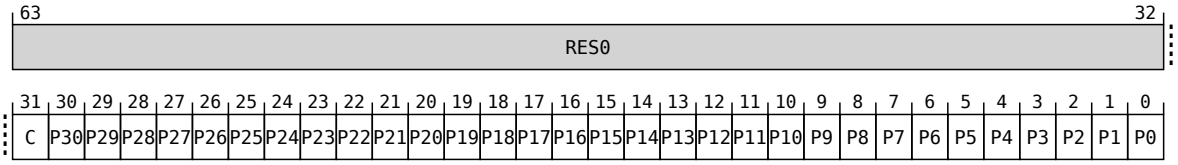
AArch64 System register PMOVSSET_EL0 bits [31:0] are architecturally mapped to External register PMOVSSET_EL0[31:0].

This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMOVSSET_EL0 are UNDEFINED.

Attributes

PMOVSSET_EL0 is a 64-bit register.

Field descriptions



Bits [63:32]

Reserved, RES0.

C, bit [31]

Unsigned overflow flag for PMCCNTR_EL0 set. On writes, allows software to set the unsigned overflow flag for PMCCNTR_EL0 to 1. On reads, returns the unsigned overflow flag for PMCCNTR_EL0 overflow status.

C	Meaning
0b0	PMCCNTR_EL0 has not overflowed.
0b1	PMCCNTR_EL0 has overflowed.

PMCR_EL0.LC controls whether an overflow is detected from unsigned overflow of PMCCNTR_EL0[31:0] or unsigned overflow of PMCCNTR_EL0[63:0].

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is W1S.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

P<m> , bits [m], for m = 30 to 0

Unsigned overflow flag for PMEVCNTR<m>_EL0 set. On writes, allows software to set the unsigned overflow flag for PMEVCNTR<m>_EL0 to 1. On reads, returns the unsigned overflow flag for PMEVCNTR<m>_EL0 overflow status.

P<m>	Meaning
0b0	PMEVCNTR<m>_EL0 has not overflowed.
0b1	PMEVCNTR<m>_EL0 has overflowed.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= GetNumEventCountersAccessible(), access to this field is RAZ/WI.
- Otherwise, access to this field is W1S.

Accessing PMOVSSET_EL0

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, PMOVSSET_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1110	0b011

```
if !IsFeatureImplemented(FEAT_PMUv3) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if PMUSERENR_EL0.EN == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = PMOVSSET_EL0;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = PMOVSSET_EL0;
elsif PSTATE.EL == EL2 then
    X[t, 64] = PMOVSSET_EL0;
```

MSR PMOVSSET_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1110	0b011

```
if !IsFeatureImplemented(FEAT_PMUv3) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if PMUSERENR_EL0.EN == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        PMOVSSET_EL0 = X[t, 64];
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        PMOVSSET_EL0 = X[t, 64];
elsif PSTATE.EL == EL2 then
    PMOVSSET_EL0 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.5.15 PMSELR_EL0, Performance Monitors Event Counter Selection Register

The PMSELR_EL0 characteristics are:

Purpose

Selects the current event counter PMEVCNTR<n>_EL0 or the cycle counter PMCCNTR_EL0.

Used in conjunction with PMXEVTYPER_EL0 to determine the event that increments a selected counter, and the modes and states in which the selected counter increments.

Used in conjunction with PMXEVCNTR_EL0 to determine the value of a selected counter.

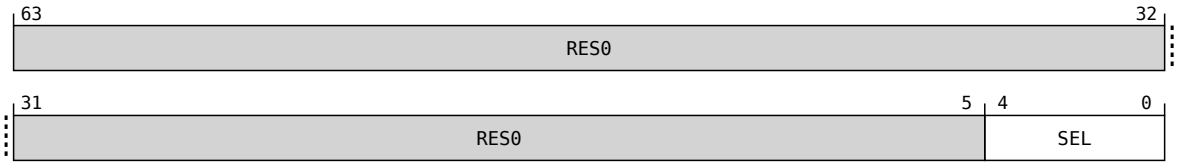
Configuration

This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMSELR_EL0 are UNDEFINED.

Attributes

PMSELR_EL0 is a 64-bit register.

Field descriptions



Bits [63:5]

Reserved, RES0.

SEL, bits [4:0]

Event counter select. Selects the counter accessed by subsequent accesses to PMXEVTYPER_EL0 and PMXEVCNTR_EL0.

SEL	Meaning
0b00000..0b11110	Select event counter PMEVCNTR<n>_EL0, where n is the value of this field: • MRS and MSR of PMXEVTYPER_EL0 access PMEVTYPER<n>_EL0. • MRS and MSR of PMXEVCNTR_EL0 access PMEVCNTR<n>_EL0.
0b11111	Select the cycle counter, PMCCNTR_EL0: • MRS and MSR of PMXEVTYPER_EL0 access PMCCFILTR_EL0. • MRS and MSR of PMXEVCNTR_EL0 are CONSTRAINED UNPREDICTABLE. For more information, see PMXEVCNTR_EL0.

If this field is set to a value greater than or equal to the number of implemented counters, but not equal to 31, then direct reads of this field return an UNKNOWN value.

For more information about the results of accesses to the event counters, including when PMSELR_EL0.SEL is set to the index of an unimplemented or inaccessible event counter, see PMXEVTYPER_EL0 and PMXEVCNTR_EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing PMSELR_EL0

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, PMSELR_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1100	0b101

```
if !IsFeatureImplemented(FEAT_PMUv3) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if PMUSERENR_EL0.<ER,EN> == '00' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = PMSELR_EL0;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = PMSELR_EL0;
elsif PSTATE.EL == EL2 then
    X[t, 64] = PMSELR_EL0;
```

MSR PMSELR_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1100	0b101

```
if !IsFeatureImplemented(FEAT_PMUv3) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if PMUSERENR_EL0.<ER,EN> == '00' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        PMSELR_EL0 = X[t, 64];
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        PMSELR_EL0 = X[t, 64];
elsif PSTATE.EL == EL2 then
    PMSELR_EL0 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.5.16 PMSWINC_EL0, Performance Monitors Software Increment Register

The PMSWINC_EL0 characteristics are:

Purpose

Increments a counter that is configured to count the Software increment event, event `0x00`. For more information, see 'SW_INCR'.

Configuration

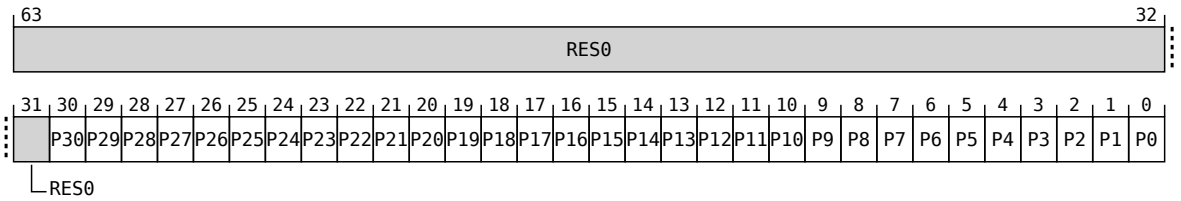
When FEAT_PMUv3_EXT32 is implemented, AArch64 System register PMSWINC_EL0 bits [31:0] are architecturally mapped to External register PMSWINC_EL0[31:0].

This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMSWINC_EL0 are UNDEFINED.

Attributes

PMSWINC_EL0 is a 64-bit register.

Field descriptions



Bits [63:31]

Reserved, RES0.

P<m> , bits [m], for m = 30 to 0

Software increment.

P<m>	Meaning
`0b0`	Write is ignored.
`0b1`	Increment PMEVCNTR<m>_EL0, if PMEVCNTR<m>_EL0 is configured to count software increment events.

Accessing this field has the following behavior:

- When m >= GetNumEventCountersAccessible(), access to this field is RAZ/WI.
- Otherwise, access to this field is WO/RAZ.

Accessing PMSWINC_EL0

Accesses to this register use the following encodings in the System register encoding space:

`MSR PMSWINC_EL0, <Xt>`

op0	op1	CRn	CRm	op2
`0b11`	`0b011`	`0b1001`	`0b1100`	`0b100`

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_PMUv3) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if PMUSERENR_EL0.<SW,EN> == '00' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        PMSWINC_EL0 = X[t, 64];
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        PMSWINC_EL0 = X[t, 64];
elsif PSTATE.EL == EL2 then
    PMSWINC_EL0 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.5.17 PMUSERENR_EL0, Performance Monitors User Enable Register

The PMUSERENR_EL0 characteristics are:

Purpose

Enables or disables EL0 access to the Performance Monitors.

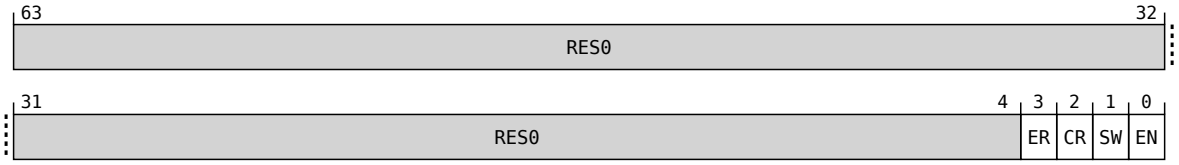
Configuration

This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMUSERENR_EL0 are UNDEFINED.

Attributes

PMUSERENR_EL0 is a 64-bit register.

Field descriptions



Bits [63:4]

Reserved, RES0.

ER, bit [3]

Event counters Read enable.

When PMUSERENR_EL0.EN is 0, PMUSERENR_EL0.ER enables EL0 reads of the event counters and EL0 reads and writes of the select register.

ER	Meaning
0b0	EL0 reads of the event counters and EL0 reads and writes of the select register are disabled, unless enabled by PMUSERENR_EL0.EN.
0b1	EL0 reads of the event counters and EL0 reads and writes of the select register are enabled, unless trapped by another control.

In AArch64 state, the register accesses affected by this control are:

- MRS reads of PMEVCNTR<n>_EL0 and PMXEVCNTR_EL0.
- MRS and MSR accesses to PMSELR_EL0.

When disabled, reads and writes generate an exception to EL1, or to EL2 when EL2 is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, and:

- AArch64 MRS and MSR accesses are reported using EC syndrome value 0x18.

This field is ignored by the PE when PMUSERENR_EL0.EN == 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

CR, bit [2]

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Cycle counter Read enable.

When PMUSERENR_EL0.EN is 0, PMUSERENR_EL0.CR enables EL0 reads of the cycle counter.

CR	Meaning
0b0	EL0 reads of the cycle counter are disabled, unless enabled by PMUSERENR_EL0.EN.
0b1	EL0 reads of the cycle counter are enabled, unless trapped by another control.

In AArch64 state, the register accesses affected by this control are:

- MRS reads of PMCCNTR_EL0.

When disabled, reads generate an exception to EL1, or to EL2 when EL2 is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, and:

- AArch64 MRS reads are reported using EC syndrome value 0x18.

This field is ignored by the PE when PMUSERENR_EL0.EN == 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SW, bit [1]

Software increment register Write enable.

When PMUSERENR_EL0.EN is 0, PMUSERENR_EL0.SW enables EL0 writes to the Software increment register.

SW	Meaning
0b0	EL0 writes to the Software increment register are disabled, unless enabled by PMUSERENR_EL0.EN.
0b1	EL0 writes to the Software increment register are enabled, unless trapped by another control.

In AArch64 state, the register accesses affected by this control are:

- MSR writes to PMSWINC_EL0.

When disabled, writes generate an exception to EL1, or to EL2 when EL2 is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, and:

- AArch64 MSR writes are reported using EC syndrome value 0x18.

This field is ignored by the PE when PMUSERENR_EL0.EN == 1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EN, bit [0]

Enable.

Enables EL0 read/write access to PMU registers.

EN	Meaning
0b0	EL0 accesses to the specified PMU System registers are trapped, unless enabled by PMUSERENR_EL0.{ER,CR,SW}.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b1	EL0 accesses to the specified PMU System registers are enabled, unless trapped by another control.

In AArch64 state, the register accesses affected by this control are:

- MRS or MSR accesses to PMCCFILTR_EL0, PMCCNTR_EL0, PMCNTENCLR_EL0, PMCNTENSET_EL0, PMCR_EL0, PMEVCNTR<n>_EL0, PMEVTYPER<n>_EL0, PMOVSCLR_EL0, PMOVSSET_EL0, PMSELR_EL0, PMXEVCNTR_EL0, and PMXEVTYPER_EL0.
- MRS reads of PMCEID0_EL0 and PMCEID1_EL0.
- MSR writes to PMSWINC_EL0.

When trapped, reads and writes generate an exception to EL1, or to EL2 when EL2 is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, and:

- AArch64 MRS and MSR accesses are reported using EC syndrome value 0x18.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PMUSERENR_EL0

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, PMUSERENR_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1110	0b000

```
if !IsFeatureImplemented(FEAT_PMUv3) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = PMUSERENR_EL0;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = PMUSERENR_EL0;
elsif PSTATE.EL == EL2 then
    X[t, 64] = PMUSERENR_EL0;
```

MSR PMUSERENR_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1110	0b000

```
if !IsFeatureImplemented(FEAT_PMUv3) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        PMUSERENR_EL0 = X[t, 64];
elsif PSTATE.EL == EL2 then
    PMUSERENR_EL0 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.5.18 PMXEVCNTR_EL0, Performance Monitors Selected Event Count Register

The PMXEVCNTR_EL0 characteristics are:

Purpose

Reads or writes the value of the selected event counter, PMEVCNTR<n>_EL0. PMSELR_EL0.SEL determines which event counter is selected.

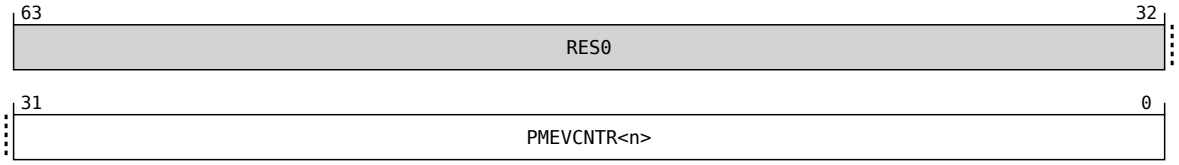
Configuration

This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMXEVCNTR_EL0 are UNDEFINED.

Attributes

PMXEVCNTR_EL0 is a 64-bit register.

Field descriptions



Bits [63:32]

Reserved, RES0.

PMEVCNTR<n>, bits [31:0]

Value of the selected event counter, PMEVCNTR<n>_EL0, where n is the value stored in PMSELR_EL0.SEL.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PMXEVCNTR_EL0

If PMSELR_EL0.SEL is greater than or equal to the number of accessible event counters, then reads and writes of PMXEVCNTR_EL0 are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP
- Accesses to the register behave as if PMSELR_EL0.SEL has an UNKNOWN value less than the number of counters accessible at the current Exception level and Security state.
- If EL2 is implemented and enabled in the current Security state, and PMSELR_EL0.SEL is less than the number of implemented event counters, accesses from EL1 or permitted accesses from EL0 are trapped to EL2.

Note

In EL0, an access is permitted if it is enabled by PMUSERENR_EL0.{ER,EN}.
If EL2 is implemented and enabled in the current Security state, in EL1 and EL0, MDCR_EL2.HPMN identifies the number of accessible event counters.
Otherwise, the number of accessible event counters is the number of implemented event counters. For more information, see MDCR_EL2.HPMN.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MRS <Xt>, PMXEVCNTR_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1101	0b010

```
if !IsFeatureImplemented(FEAT_PMUv3) then
    UNDEFINED;
elsif UInt(PMSELR_EL0.SEL) >= GetNumEventCountersSelfHosted() then
    ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
elsif PSTATE.EL == EL0 then
    if PMUSERENR_EL0.<ER,EN> == '00' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif UInt(PMSELR_EL0.SEL) >= GetNumEventCountersAccessible() then
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    else
        X[t, 64] = PMEVCNTR_EL0[UInt(PMSELR_EL0.SEL)];
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif UInt(PMSELR_EL0.SEL) >= GetNumEventCountersAccessible() then
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    else
        X[t, 64] = PMEVCNTR_EL0[UInt(PMSELR_EL0.SEL)];
elsif PSTATE.EL == EL2 then
    X[t, 64] = PMEVCNTR_EL0[UInt(PMSELR_EL0.SEL)];
```

MSR PMXEVCNTR_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1101	0b010

```
if !IsFeatureImplemented(FEAT_PMUv3) then
    UNDEFINED;
elsif UInt(PMSELR_EL0.SEL) >= GetNumEventCountersSelfHosted() then
    ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
elsif PSTATE.EL == EL0 then
    if PMUSERENR_EL0.EN == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif UInt(PMSELR_EL0.SEL) >= GetNumEventCountersAccessible() then
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    else
        PMEVCNTR_EL0[UInt(PMSELR_EL0.SEL)] = X[t, 64];
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif UInt(PMSELR_EL0.SEL) >= GetNumEventCountersAccessible() then
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    else
        PMEVCNTR_EL0[UInt(PMSELR_EL0.SEL)] = X[t, 64];
elsif PSTATE.EL == EL2 then
    PMEVCNTR_EL0[UInt(PMSELR_EL0.SEL)] = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.5.19 PMXEVTYPER_EL0, Performance Monitors Selected Event Type Register

The PMXEVTYPER_EL0 characteristics are:

Purpose

When PMSELR_EL0.SEL selects an event counter, this accesses a PMEVTYPER<n>_EL0 register. When PMSELR_EL0.SEL selects the cycle counter, this accesses PMCCFILTR_EL0.

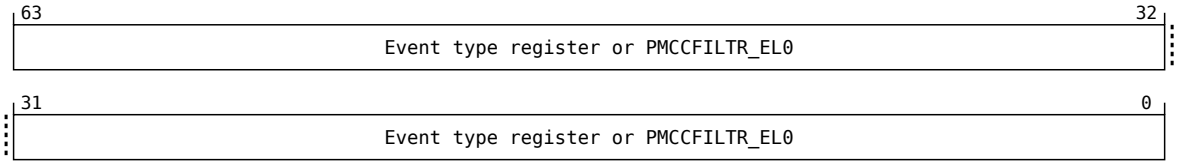
Configuration

This register is present only when FEAT_PMUv3 is implemented. Otherwise, direct accesses to PMXEVTYPER_EL0 are UNDEFINED.

Attributes

PMXEVTYPER_EL0 is a 64-bit register.

Field descriptions



EVTYPERn, bits [63:0]

When PMSELR_EL0.SEL == 31, this register accesses PMCCFILTR_EL0.

Otherwise, this register accesses PMEVTYPER<n>_EL0 where n is the value in PMSELR_EL0.SEL.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PMXEVTYPER_EL0

If PMSELR_EL0.SEL is not 31, and is greater than or equal to the number of accessible event counters, then reads and writes of PMXEVTYPER_EL0 are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP.
- Accesses to the register behave as if PMSELR_EL0.SEL has an UNKNOWN value less than the number of event counters accessible at the current Exception level and Security state.
- Accesses to the register behave as if PMSELR_EL0.SEL is 31.
- If PMSELR_EL0 is less than the number of implemented event counters, accesses from EL1 or permitted accesses from EL0 are trapped to EL2.

Note

In EL0, an access is permitted if it is enabled by PMUSERENR_EL0.EN.
In in EL1 and EL0, MDCR_EL2.HPMN identifies the number of accessible event counters.
Otherwise, the number of accessible event counters is the number of implemented event counters. For more information, see MDCR_EL2.HPMN.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PMXEVTYPER_EL0
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1101	0b001

```
if !IsFeatureImplemented(FEAT_PMUv3) then
    UNDEFINED;
elsif UInt(PMSELR_EL0.SEL) != 31 && UInt(PMSELR_EL0.SEL) >= GetNumEventCountersSelfHosted() then
    ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
elsif PSTATE.EL == EL0 then
    if PMUSERENR_EL0.EN == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif UInt(PMSELR_EL0.SEL) != 31 && UInt(PMSELR_EL0.SEL) >= GetNumEventCountersAccessible() then
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    elsif UInt(PMSELR_EL0.SEL) == 31 then
        X[t, 64] = PMCCFILTR_EL0;
    else
        X[t, 64] = PMEVTYPER_EL0[UInt(PMSELR_EL0.SEL)];
elsif PSTATE.EL == EL1 then
    if MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif UInt(PMSELR_EL0.SEL) != 31 && UInt(PMSELR_EL0.SEL) >= GetNumEventCountersAccessible() then
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    elsif UInt(PMSELR_EL0.SEL) == 31 then
        X[t, 64] = PMCCFILTR_EL0;
    else
        X[t, 64] = PMEVTYPER_EL0[UInt(PMSELR_EL0.SEL)];
elsif PSTATE.EL == EL2 then
    if UInt(PMSELR_EL0.SEL) == 31 then
        X[t, 64] = PMCCFILTR_EL0;
    else
        X[t, 64] = PMEVTYPER_EL0[UInt(PMSELR_EL0.SEL)];
```

MSR PMXEVTYPER_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1001	0b1101	0b001

```
if !IsFeatureImplemented(FEAT_PMUv3) then
    UNDEFINED;
elsif UInt(PMSELR_EL0.SEL) != 31 && UInt(PMSELR_EL0.SEL) >= GetNumEventCountersSelfHosted() then
    ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
elsif PSTATE.EL == EL0 then
    if PMUSERENR_EL0.EN == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif UInt(PMSELR_EL0.SEL) != 31 && UInt(PMSELR_EL0.SEL) >= GetNumEventCountersAccessible() then
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    elsif UInt(PMSELR_EL0.SEL) == 31 then
        PMCCFILTR_EL0 = X[t, 64];
    else
        PMEVTYPER_EL0[UInt(PMSELR_EL0.SEL)] = X[t, 64];
elsif PSTATE.EL == EL1 then
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if MDCR_EL2.TPM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif UInt(PMSELR_EL0.SEL) != 31 && UInt(PMSELR_EL0.SEL) >= GetNumEventCountersAccessible() then
        ConstrainUnpredictableProcedure(Unpredictable_PMUEVENTCOUNTER);
    elsif UInt(PMSELR_EL0.SEL) == 31 then
        PMCCFILTR_EL0 = X[t, 64];
    else
        PMEVTYPER_EL0[UInt(PMSELR_EL0.SEL)] = X[t, 64];
elsif PSTATE.EL == EL2 then
    if UInt(PMSELR_EL0.SEL) == 31 then
        PMCCFILTR_EL0 = X[t, 64];
    else
        PMEVTYPER_EL0[UInt(PMSELR_EL0.SEL)] = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.6 PMSA registers

This section lists the PMSA registers in AArch64.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.6.1 MPUIR_EL1, MPU Type Register (EL1)

The MPUIR_EL1 characteristics are:

Purpose

Identifies the number of regions supported by the EL1 MPU.

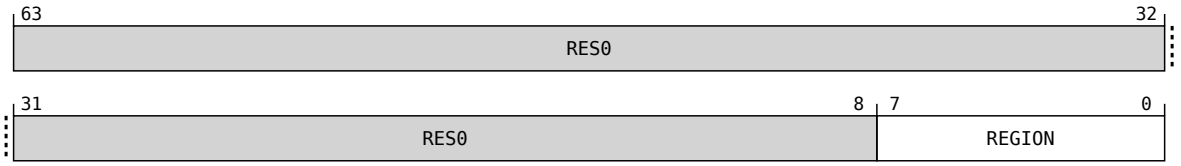
Configuration

There are no configuration notes.

Attributes

MPUIR_EL1 is a 64-bit register.

Field descriptions



Bits [63:8]

Reserved, RES0.

REGION, bits [7:0]

The number of EL1 MPU regions supported.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing MPUIR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, MPUIR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0000	0b100

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_IDST) then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TID1 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif VTCR_EL2.MSA == '1' then
        UNDEFINED;
    else
        X[t, 64] = MPUIR_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = MPUIR_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.6.2 MPUIR_EL2, MPU Type Register (EL2)

The MPUIR_EL2 characteristics are:

Purpose

Identifies the number of regions supported by the EL2 MPU.

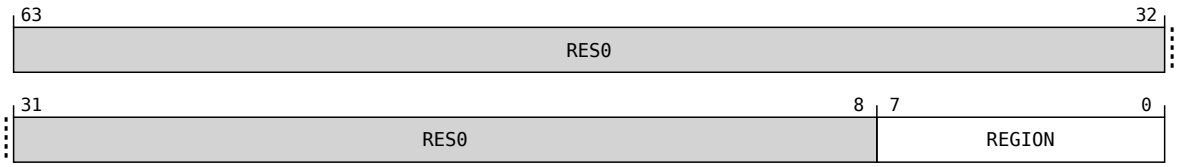
Configuration

There are no configuration notes.

Attributes

MPUIR_EL2 is a 64-bit register.

Field descriptions



Bits [63:8]

Reserved, RES0.

REGION, bits [7:0]

The number of EL2 MPU regions supported.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing MPUIR_EL2

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, MPUIR_EL2`

op0	op1	CRn	CRm	op2
0b11	0b100	0b0000	0b0000	0b100

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = MPUIR_EL2;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.6.3 PRBAR_EL1, Protection Region Base Address Register (EL1)

The PRBAR_EL1 characteristics are:

Purpose

Provides access to the base addresses for the EL1 MPU region. PRSELR_EL1.REGION determines which MPU region is selected.

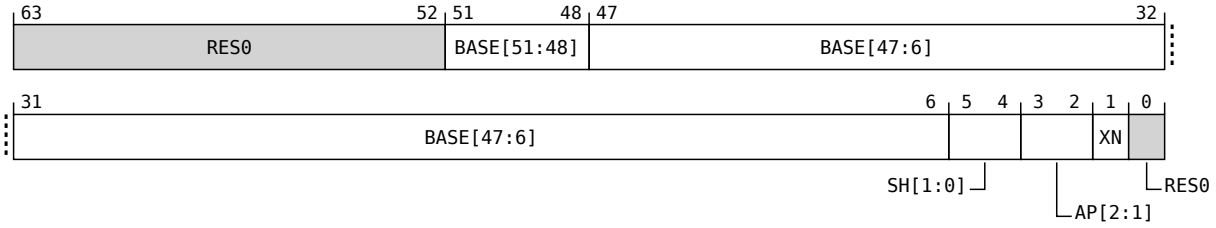
Configuration

All bits above implemented physical address range in this register should be treated as RES0.

Attributes

PRBAR_EL1 is a 64-bit register.

Field descriptions



Bits [63:52]

Reserved, RES0.

BASE[51:48], bits [51:48]

When FEAT_LPA is implemented:

Extension to BASE[47:6]. When FEAT_LPA is implemented, BASE[51:48] form the upper part of the address value. Otherwise, for implementations with fewer than 52 physical address bits, the upper bits of this field, corresponding to address bits that are not implemented, are RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

BASE[47:6], bits [47:6]

Bits[47:6] of the lower inclusive limit of the selected EL1 MPU memory region. This value is zero extended to provide the base address to be checked against.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SH[1:0], bits [5:4]

Shareability attribute.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SH[1:0]	Meaning
0b00	Non-shareable
0b01	Reserved, CONSTRAINED UNPREDICTABLE
0b10	Outer Shareable
0b11	Inner Shareable

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

AP[2:1], bits [3:2]

Access Permission attributes.

AP[2:1]	Meaning
0b00	Read/write at EL1, no access at EL0
0b01	Read/write at EL1 and EL0
0b10	Read-only at EL1, no access at EL0
0b11	Read-only at EL1 and EL0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

XN, bit [1]

Execute Never

XN	Meaning
0b0	Execution of instructions fetched from the region is permitted.
0b1	Execution of instructions fetched from the region is not permitted.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [0]

Reserved, RES0.

Accessing PRBAR_EL1

Any access to MPU region register PRBAR_EL1 above the number of implemented regions specified by MPUIR_EL1.REGION is CONSTRAINED UNPREDICTABLE.

CONSTRAINED UNPREDICTABLE behavior is defined as:

- Reads of unimplemented PRBAR_EL1 register return an UNKNOWN value.
- Writes to unimplemented PRBAR_EL1 register make all PRBAR_EL1 registers value UNKNOWN.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PRBAR_EL1
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b000	0b0110	0b1000	0b000

```
if UInt(PRSELR_EL1.REGION) >= UInt(MPUIR_EL1.REGION) then
    ConstrainUnpredictableProcedure(Unpredictable_UnimplementedMPURegion_RW_UNKNOWN);
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif VTCR_EL2.MSA == '1' then
        UNDEFINED;
    else
        X[t, 64] = PRBAR_EL1[UInt(PRSELR_EL1.REGION)];
elsif PSTATE.EL == EL2 then
    X[t, 64] = PRBAR_EL1[UInt(PRSELR_EL1.REGION)];
```

MSR PRBAR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0110	0b1000	0b000

```
if UInt(PRSELR_EL1.REGION) >= UInt(MPUIR_EL1.REGION) then
    ConstrainUnpredictableProcedure(Unpredictable_UnimplementedMPURegion_RW_UNKNOWN);
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif VTCR_EL2.MSA == '1' then
        UNDEFINED;
    else
        PRBAR_EL1[UInt(PRSELR_EL1.REGION)] = X[t, 64];
elsif PSTATE.EL == EL2 then
    PRBAR_EL1[UInt(PRSELR_EL1.REGION)] = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.6.4 PRBAR_EL2, Protection Region Base Address Register (EL2)

The PRBAR_EL2 characteristics are:

Purpose

Provides access to the base addresses for the EL2 MPU region. PRSELR_EL2.REGION determines which MPU region is selected.

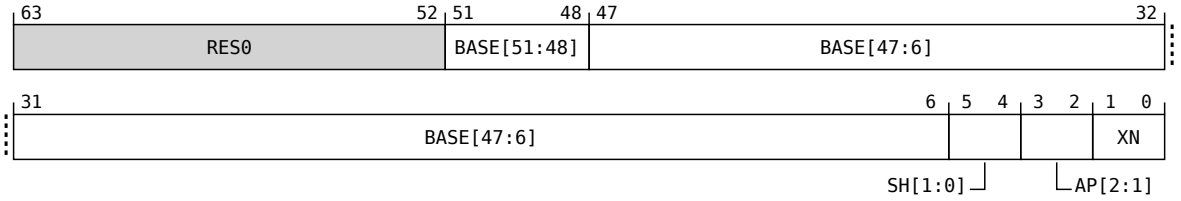
Configuration

All bits above implemented physical address range in this register should be treated as RES0.

Attributes

PRBAR_EL2 is a 64-bit register.

Field descriptions



Bits [63:52]

Reserved, RES0.

BASE[51:48], bits [51:48]

When FEAT_LPA is implemented:

Extension to BASE[47:6]. When FEAT_LPA is implemented, BASE[51:48] form the upper part of the address value. Otherwise, for implementations with fewer than 52 physical address bits, the upper bits of this field, corresponding to address bits that are not implemented, are RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

BASE[47:6], bits [47:6]

Bits[47:6] of the lower inclusive limit of the selected EL2 MPU memory region. This value is zero extended to provide the base address to be checked against.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

SH[1:0], bits [5:4]

Shareability attribute.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SH[1:0]	Meaning
0b00	Non-shareable
0b01	Reserved, CONSTRAINED UNPREDICTABLE
0b10	Outer Shareable
0b11	Inner Shareable

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

AP[2:1], bits [3:2]

Access Permission attributes.

AP[2:1]	Meaning
0b00	Read/write at EL2, no access at EL1 or EL0
0b01	Read/write at EL2, EL1 and EL0
0b10	Read-only at EL2, no access at EL1 or EL0
0b11	Read-only at EL2, EL1 and EL0

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

XN, bits [1:0]

Execute Never. For

- Stage 1 EL2 translation regime and
- Stage 2 EL1&0 translation regime when FEAT_XNX is not implemented

XN[1] determines whether execution of the instructions fetched from the MPU memory region is permitted. In this case, XN[0] is RES0

For stage 2 EL1&0 translation regime when FEAT_XNX is implemented, the behavior of XN[1:0] is same as that defined by VMSAv8-64 for EL1&0 stage 2 translation table XN[1:0],bits[54:53] field in Armv8-A architecture.

XN	Meaning
0b00	Execution of instructions fetched from the region is permitted.
0b01	Execution of instructions fetched from the region is not permitted.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PRBAR_EL2

Any access to MPU region register PRBAR_EL2 above the number of implemented regions specified by MPUIR_EL2.REGION is CONSTRAINED UNPREDICTABLE.

CONSTRAINED UNPREDICTABLE behavior is defined as:

- Reads of unimplemented PRBAR_EL2 register return an UNKNOWN value.
- Writes to unimplemented PRBAR_EL2 register make all PRBAR_EL2 registers value UNKNOWN.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MRS <Xt>, PRBAR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0110	0b1000	0b000

```
if UInt(PRSELR_EL2.REGION) >= UInt(MPUIR_EL2.REGION) then
    ConstrainUnpredictableProcedure(Unpredictable_UnimplementedMPURegion_RW_UNKNOWN);
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = PRBAR_EL2[UInt(PRSELR_EL2.REGION)];
```

MSR PRBAR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0110	0b1000	0b000

```
if UInt(PRSELR_EL2.REGION) >= UInt(MPUIR_EL2.REGION) then
    ConstrainUnpredictableProcedure(Unpredictable_UnimplementedMPURegion_RW_UNKNOWN);
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    PRBAR_EL2[UInt(PRSELR_EL2.REGION)] = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.6.5 PRENR_EL1, Protection Region Enable Register (EL1)

The PRENR_EL1 characteristics are:

Purpose

Provides direct access to the PRLAR_EL1.EN bits of EL1 MPU regions from 0 to 31.

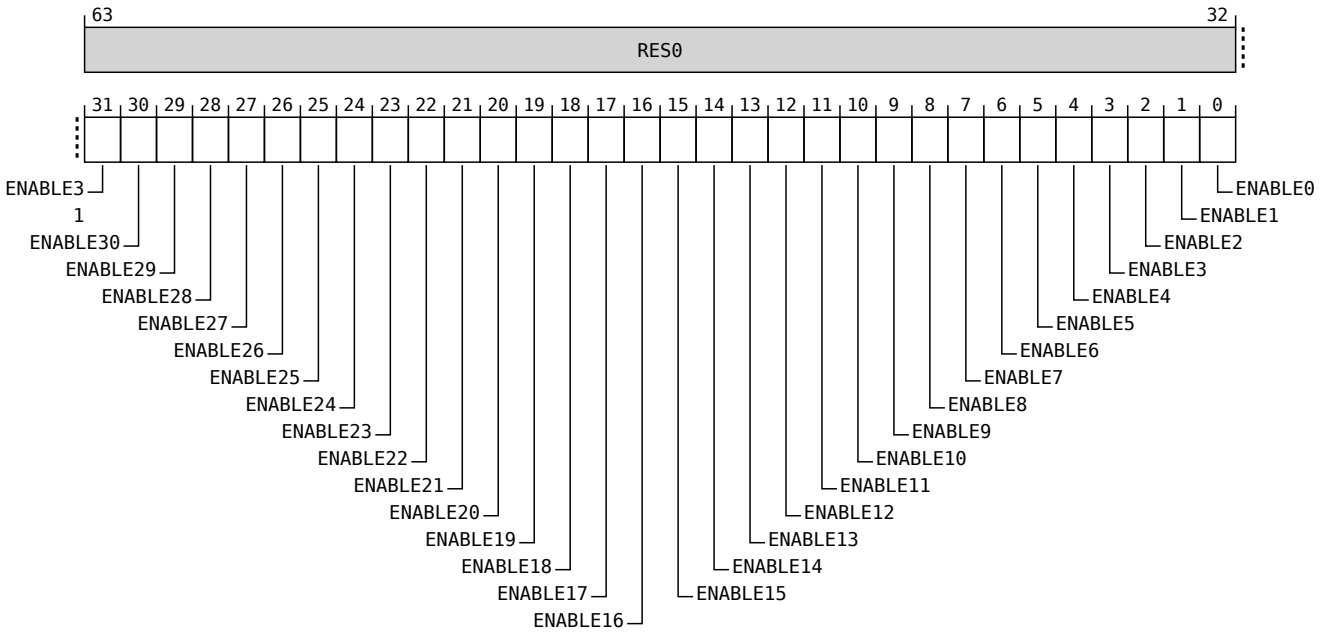
Configuration

There are no configuration notes.

Attributes

PRENR_EL1 is a 64-bit register.

Field descriptions



Bits [63:32]

Reserved, RES0.

ENABLE<n> , bits [n], for n = 31 to 0

Enable bit. Each bit, n, enables or disables the respective EL1 MPU region. The bits associated with the unimplemented MPU regions are RAZ/WI.

ENABLE<n>	Meaning
0b0	Disables the EL1 MPU n region.
0b1	Enables the EL1 MPU n region.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing PRENR_EL1

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, PRENR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0110	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif VTCR_EL2.MSA == '1' then
        UNDEFINED;
    else
        X[t, 64] = PRENR_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = PRENR_EL1;
```

MSR PRENR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0110	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif VTCR_EL2.MSA == '1' then
        UNDEFINED;
    else
        PRENR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    PRENR_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.6.6 PRENR_EL2, Protection Region Enable Register (EL2)

The PRENR_EL2 characteristics are:

Purpose

Provides direct access to the PRLAR_EL2.EN bits of EL2 MPU regions from 0 to 31.

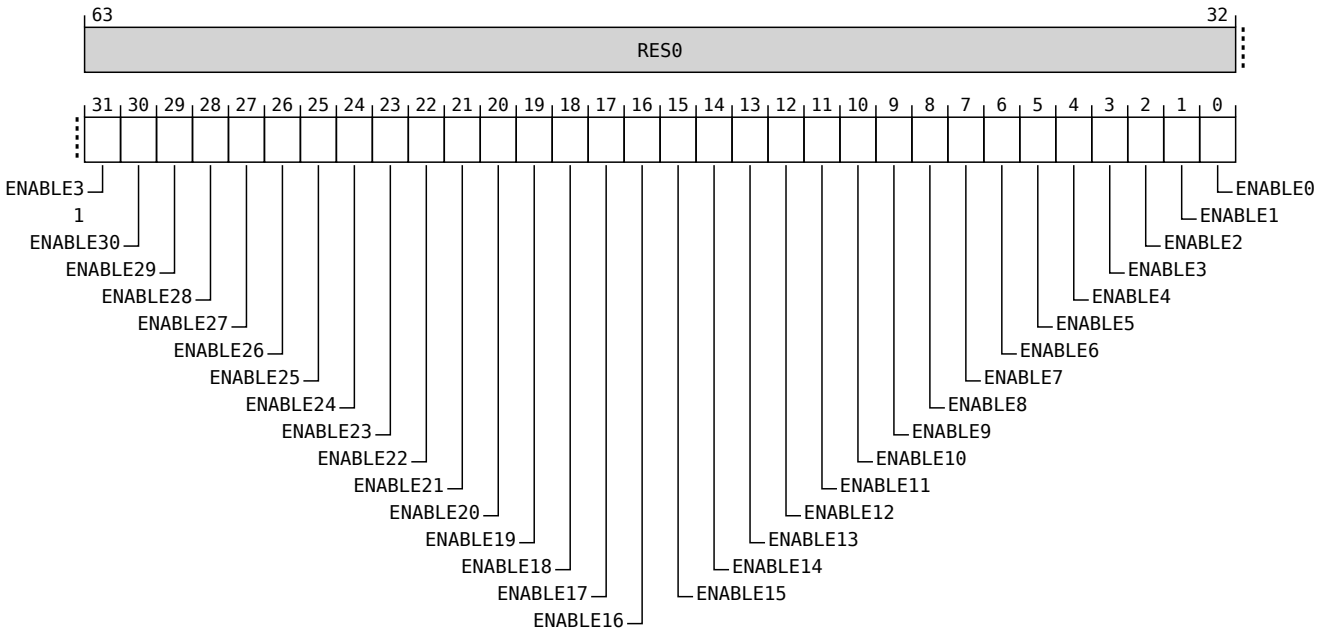
Configuration

There are no configuration notes.

Attributes

PRENR_EL2 is a 64-bit register.

Field descriptions



Bits [63:32]

Reserved, RES0.

ENABLE<n> , bits [n], for n = 31 to 0

Enable bit. Each bit, n, enables or disables the respective EL2 MPU region. The bits associated with the unimplemented MPU regions are RAZ/WI.

ENABLE<n>	Meaning
0b0	Disables the EL2 MPU n region.
0b1	Enables the EL2 MPU n region.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing PRENR_EL2

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, PRENR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0110	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = PRENR_EL2;
```

MSR PRENR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0110	0b0001	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    PRENR_EL2 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.6.7 PRLAR_EL1, Protection Region Limit Address Register (EL1)

The PRLAR_EL1 characteristics are:

Purpose

Provides access to the limit addresses for the EL1 MPU region. PRSELR_EL1.REGION determines which MPU region is selected.

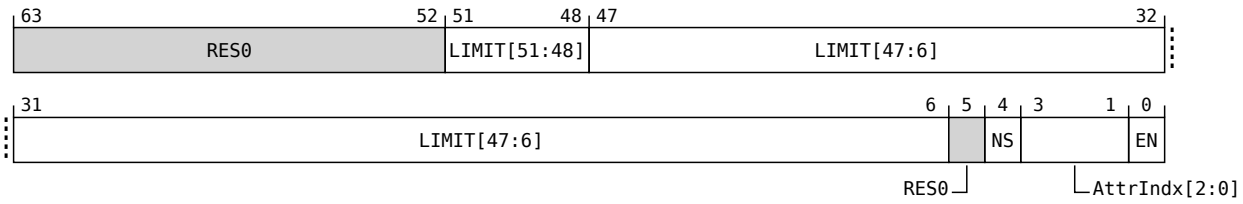
Configuration

All bits above implemented physical address range in this register should be treated as RES0.

Attributes

PRLAR_EL1 is a 64-bit register.

Field descriptions



Bits [63:52]

Reserved, RES0.

LIMIT[51:48], bits [51:48]

When FEAT_LPA is implemented:

Extension to LIMIT[47:6]. When FEAT_LPA is implemented, LIMIT[51:48] form the upper part of the address value. Otherwise, for implementations with fewer than 52 physical address bits, the upper bits of this field, corresponding to address bits that are not implemented, are RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

LIMIT[47:6], bits [47:6]

Bits[47:6] of the upper inclusive limit of the selected EL1 MPU memory region. This value is concatenated with the value `0x3F` to provide the limit address to be checked against.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [5]

Reserved, RES0.

NS, bit [4]

Non-secure bit. Specifies whether the output address is in the Secure or Non-secure memory.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NS	Meaning
0b0	Output address is in Secure address space.
0b1	Output address is in Non-secure address space.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

AttrIndx[2:0], bits [3:1]

Selects attributes from within the associated Memory Attribute Indirection Register.

AttrIndx[2:0]	Meaning
0b000	Select the Attr0 field from MAIR_EL1.
0b001	Select the Attr1 field from MAIR_EL1.
0b010	Select the Attr2 field from MAIR_EL1.
0b011	Select the Attr3 field from MAIR_EL1.
0b100	Select the Attr4 field from MAIR_EL1.
0b101	Select the Attr5 field from MAIR_EL1.
0b110	Select the Attr6 field from MAIR_EL1.
0b111	Select the Attr7 field from MAIR_EL1.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EN, bit [0]

Region enable bit.

EN	Meaning
0b0	Region disabled.
0b1	Region enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Accessing PRLAR_EL1

Any access to MPU region register PRLAR_EL1 above the number of implemented regions specified by MPUIR_EL1.REGION is CONSTRAINED UNPREDICTABLE.

CONSTRAINED UNPREDICTABLE behavior is defined as:

- Reads of unimplemented PRLAR_EL1 register return an UNKNOWN value.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Writes to unimplemented PRLAR_EL1 register make all PRLAR_EL1 registers value UNKNOWN.

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, PRLAR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0110	0b1000	0b001

```
if UInt(PRSELR_EL1.REGION) >= UInt(MPUIR_EL1.REGION) then
    ConstrainUnpredictableProcedure(Unpredictable_UnimplementedMPURegion_RW_UNKNOWN);
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif VTCR_EL2.MSA == '1' then
        UNDEFINED;
    else
        X[t, 64] = PRLAR_EL1[UInt(PRSELR_EL1.REGION)];
elsif PSTATE.EL == EL2 then
    X[t, 64] = PRLAR_EL1[UInt(PRSELR_EL1.REGION)];
```

MSR PRLAR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0110	0b1000	0b001

```
if UInt(PRSELR_EL1.REGION) >= UInt(MPUIR_EL1.REGION) then
    ConstrainUnpredictableProcedure(Unpredictable_UnimplementedMPURegion_RW_UNKNOWN);
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif VTCR_EL2.MSA == '1' then
        UNDEFINED;
    else
        PRLAR_EL1[UInt(PRSELR_EL1.REGION)] = X[t, 64];
elsif PSTATE.EL == EL2 then
    PRLAR_EL1[UInt(PRSELR_EL1.REGION)] = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.6.8 PRLAR_EL2, Protection Region Limit Address Register (EL2)

The PRLAR_EL2 characteristics are:

Purpose

Provides access to the limit addresses for the EL2 MPU region. PRSELR_EL2.REGION determines which MPU region is selected.

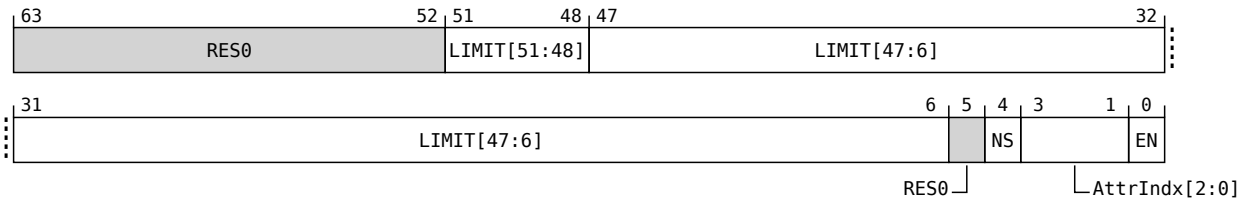
Configuration

All bits above implemented physical address range in this register should be treated as RES0.

Attributes

PRLAR_EL2 is a 64-bit register.

Field descriptions



Bits [63:52]

Reserved, RES0.

LIMIT[51:48], bits [51:48]

When FEAT_LPA is implemented:

Extension to LIMIT[47:6]. When FEAT_LPA is implemented, LIMIT[51:48] form the upper part of the address value. Otherwise, for implementations with fewer than 52 physical address bits, the upper bits of this field, corresponding to address bits that are not implemented, are RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

LIMIT[47:6], bits [47:6]

Bits[47:6] of the upper inclusive limit of the selected EL2 MPU memory region. This value is concatenated with the value `0x3F` to provide the limit address to be checked against.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [5]

Reserved, RES0.

NS, bit [4]

Non-secure bit. Specifies whether the output address is in the Secure or Non-secure memory.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NS	Meaning
0b0	Output address is in Secure address space.
0b1	Output address is in Non-secure address space.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

AttrIndx[2:0], bits [3:1]

Selects attributes from within the associated Memory Attribute Indirection Register.

AttrIndx[2:0]	Meaning
0b000	Select the Attr0 field from MAIR_EL2.
0b001	Select the Attr1 field from MAIR_EL2.
0b010	Select the Attr2 field from MAIR_EL2.
0b011	Select the Attr3 field from MAIR_EL2.
0b100	Select the Attr4 field from MAIR_EL2.
0b101	Select the Attr5 field from MAIR_EL2.
0b110	Select the Attr6 field from MAIR_EL2.
0b111	Select the Attr7 field from MAIR_EL2.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EN, bit [0]

Region enable bit.

EN	Meaning
0b0	Region disabled.
0b1	Region enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Accessing PRLAR_EL2

Any access to MPU region register PRLAR_EL2 above the number of implemented regions specified by MPUIR_EL2.REGION is CONSTRAINED UNPREDICTABLE.

CONSTRAINED UNPREDICTABLE behavior is defined as:

- Reads of unimplemented PRLAR_EL2 register return an UNKNOWN value.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Writes to unimplemented PRLAR_EL2 register make all PRLAR_EL2 registers value UNKNOWN.

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, PRLAR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b0110	0b1000	0b001

```
if UInt(PRSELR_EL2.REGION) >= UInt(MPUIR_EL2.REGION) then
    ConstrainUnpredictableProcedure(Unpredictable_UnimplementedMPURegion_RW_UNKNOWN);
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = PRLAR_EL2[UInt(PRSELR_EL2.REGION)];
```

MSR PRLAR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0110	0b1000	0b001

```
if UInt(PRSELR_EL2.REGION) >= UInt(MPUIR_EL2.REGION) then
    ConstrainUnpredictableProcedure(Unpredictable_UnimplementedMPURegion_RW_UNKNOWN);
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    PRLAR_EL2[UInt(PRSELR_EL2.REGION)] = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.6.9 PRSELR_EL1, Protection Region Selection Register (EL1)

The PRSELR_EL1 characteristics are:

Purpose

Selects the region number for the EL1 MPU region associated with the PRBAR_EL1 and PRLAR_EL1 registers.

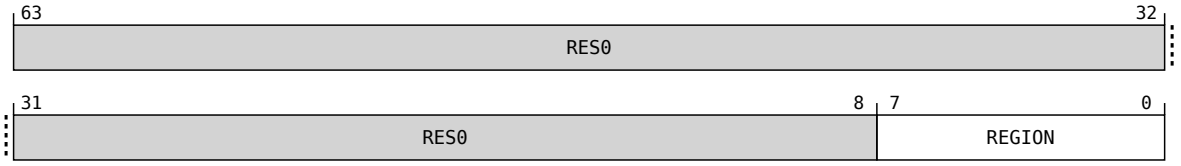
Configuration

There are no configuration notes.

Attributes

PRSELR_EL1 is a 64-bit register.

Field descriptions



Bits [63:8]

Reserved, RES0.

REGION, bits [7:0]

The number of the current EL1 MPU region visible in PRBAR_EL1 and PRLAR_EL1. For N implemented MPU regions, memory region numbering starts at 0 and increments by 1 to the value N-1.

Writing a value greater than or equal to the number of implemented MPU regions specified by MPUIR_EL1.REGION, results in CONSTRAINED UNPREDICTABLE behavior.

CONSTRAINED UNPREDICTABLE behavior is that PRSELR_EL1 register becomes UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PRSELR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PRSELR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0110	0b0010	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif VTCR_EL2.MSA == '1' then
        UNDEFINED;
    else
        X[t, 64] = PRSELR_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = PRSELR_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR PRSELR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0110	0b0010	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif VTCR_EL2.MSA == '1' then
        UNDEFINED;
    else
        PRSELR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    PRSELR_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.6.10 PRSELR_EL2, Protection Region Selection Register (EL2)

The PRSELR_EL2 characteristics are:

Purpose

Selects the region number for the EL2 MPU region associated with the PRBAR_EL2 and PRLAR_EL2 registers.

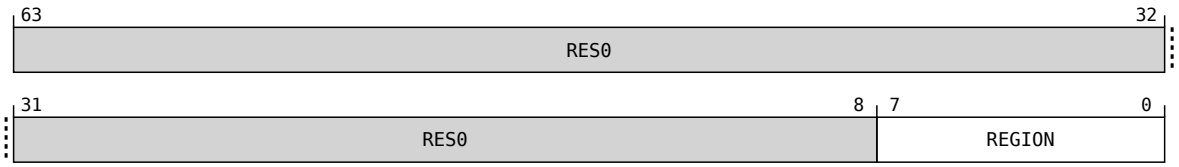
Configuration

There are no configuration notes.

Attributes

PRSELR_EL2 is a 64-bit register.

Field descriptions



Bits [63:8]

Reserved, RES0.

REGION, bits [7:0]

The number of the current EL2 MPU region visible in PRBAR_EL2 and PRLAR_EL2. For N implemented MPU regions, memory region numbering starts at 0 and increments by 1 to the value N-1.

Writing a value greater than or equal to the number of implemented MPU regions specified by MPUIR_EL2.REGION, results in CONSTRAINED UNPREDICTABLE behavior.

CONSTRAINED UNPREDICTABLE behavior is that PRSELR_EL2 register becomes UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PRSELR_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, PRSELR_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b0110	0b0010	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = PRSELR_EL2;
```

```
MSR PRSELR_EL2, <Xt>
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b100	0b0110	0b0010	0b001

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    PRSELR_EL2 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.7 RAS registers

This section lists RAS registers in AArch64.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.7.1 DISR_EL1, Deferred Interrupt Status Register

The DISR_EL1 characteristics are:

Purpose

Records that an SError exception has been consumed by an ESB instruction.

Configuration

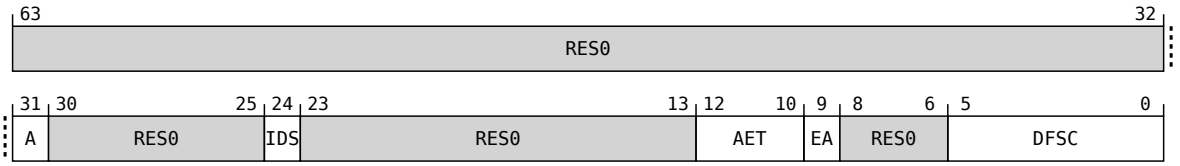
This register is present only when FEAT_RAS is implemented. Otherwise, direct accesses to DISR_EL1 are UNDEFINED.

Attributes

DISR_EL1 is a 64-bit register.

Field descriptions

When DISR_EL1.IDS == '0':



Bits [63:32]

Reserved, RES0.

A, bit [31]

Set to 1 when an ESB instruction defers an asynchronous SError exception. If the implementation does not include any sources of SError exception that can be synchronized by an Error Synchronization Barrier, then this bit is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [30:25]

Reserved, RES0.

IDS, bit [24]

Indicates the deferred SError exception type.

IDS	Meaning
0b0	Deferred error uses architecturally-defined format.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [23:13]

Reserved, RES0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

AET, bits [12:10]

Asynchronous Error Type. See the description of ESR_ELx.AET for an SError exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EA, bit [9]

External abort Type. See the description of ESR_ELx.EA for an SError exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [8:6]

Reserved, RES0.

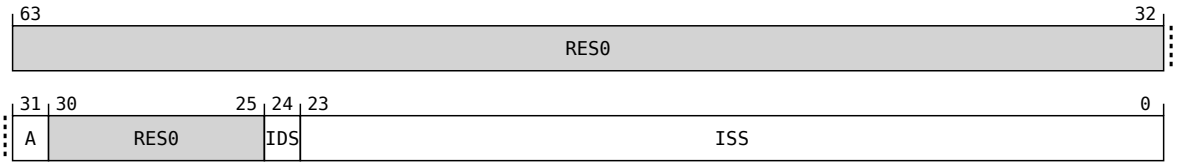
DFSC, bits [5:0]

Fault Status Code. See the description of ESR_ELx.DFSC for an SError exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

When DISR_EL1.IDS == '1':



Bits [63:32]

Reserved, RES0.

A, bit [31]

Set to 1 when an ESB instruction defers an asynchronous SError exception. If the implementation does not include any sources of SError exception that can be synchronized by an Error Synchronization Barrier, then this bit is RES0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [30:25]

Reserved, RES0.

IDS, bit [24]

Indicates the deferred SError exception type.

IDS	Meaning
0b1	Deferred error uses IMPLEMENTATION DEFINED format.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IMPLEMENTATION DEFINED, bits [23:0]

IMPLEMENTATION DEFINED syndrome. See the description of ESR_ELx[23:0] for an SError exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing DISR_EL1

An indirect write to DISR_EL1 made by an ESB instruction does not require an explicit synchronization operation for the value that is written to be observed by a direct read of DISR_EL1 occurring in program order after the ESB instruction.

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, DISR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1100	0b0001	0b001

```
if !IsFeatureImplemented(FEAT_RAS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.AMO == '1' then
        X[t, 64] = VDISR_EL2;
    else
        X[t, 64] = DISR_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = DISR_EL1;
```

MSR DISR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b1100	0b0001	0b001

```
if !IsFeatureImplemented(FEAT_RAS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.AMO == '1' then
        VDISR_EL2 = X[t, 64];
    else
        DISR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    DISR_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.7.2 ERRIDR_EL1, Error Record ID Register

The ERRIDR_EL1 characteristics are:

Purpose

Defines the highest numbered index of the error records that can be accessed through the Error Record System registers.

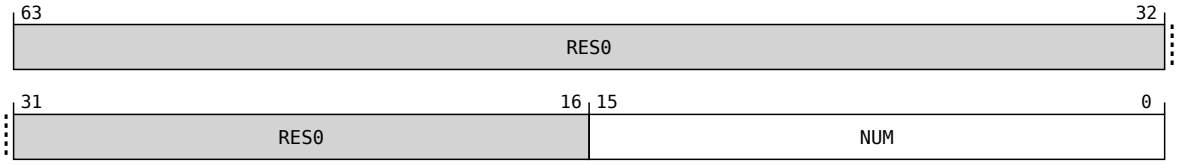
Configuration

This register is present only when FEAT_RAS is implemented. Otherwise, direct accesses to ERRIDR_EL1 are UNDEFINED.

Attributes

ERRIDR_EL1 is a 64-bit register.

Field descriptions



Bits [63:16]

Reserved, RES0.

NUM, bits [15:0]

Highest numbered index of the records that can be accessed through the Error Record System registers plus one. Zero indicates no records can be accessed through the Error Record System registers.

Each implemented record is owned by a node. A node might own multiple records.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing ERRIDR_EL1

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, ERRIDR_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0011	0b000

```
if !IsFeatureImplemented(FEAT_RAS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TERR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ERRIDR_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ERRIDR_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.7.3 ERRSELR_EL1, Error Record Select Register

The ERRSELR_EL1 characteristics are:

Purpose

Selects an error record to be accessed through the Error Record System registers.

Configuration

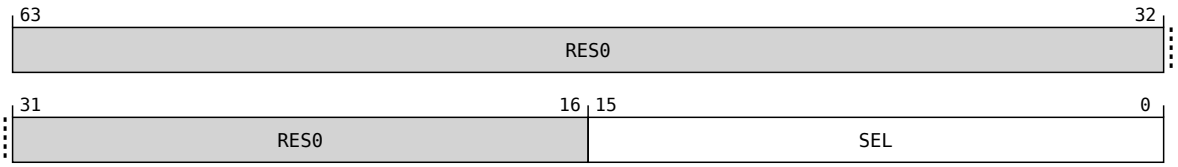
If ERRIDR_EL1 indicates that zero error records are implemented, then it is IMPLEMENTATION DEFINED whether ERRSELR_EL1 is UNDEFINED or RES0.

This register is present only when FEAT_RAS is implemented. Otherwise, direct accesses to ERRSELR_EL1 are UNDEFINED.

Attributes

ERRSELR_EL1 is a 64-bit register.

Field descriptions



Bits [63:16]

Reserved, RES0.

SEL, bits [15:0]

Selects the error record accessed through the ERX registers.

For example, if ERRSELR_EL1.SEL is 0x0004, then direct reads and writes of ERXSTATUS_EL1 access ERR4STATUS.

If ERRSELR_EL1.SEL is greater than or equal to ERRIDR_EL1.NUM, then all of the following apply:

- The value read back from ERRSELR_EL1.SEL is UNKNOWN.
- Unless the access to an ERX*_EL1 register generates a higher priority exception, one of the following occurs:
  - An UNKNOWN error record is selected.
  - The ERX*_EL1 registers are RAZ/WI.
  - ERX*_EL1 register reads and writes are NOPs.
  - ERX*_EL1 register reads and writes are UNDEFINED.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing ERRSELR_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ERRSELR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0011	0b001

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if !IsFeatureImplemented(FEAT_RAS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TERR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ERRSELR_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ERRSELR_EL1;
```

MSR ERRSELR_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0011	0b001

```
if !IsFeatureImplemented(FEAT_RAS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TERR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        ERRSELR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    ERRSELR_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.7.4 ERXADDR_EL1, Selected Error Record Address Register

The ERXADDR_EL1 characteristics are:

Purpose

Accesses ERR<n>ADDR for the error record <n> selected by ERRSELR_EL1.SEL.

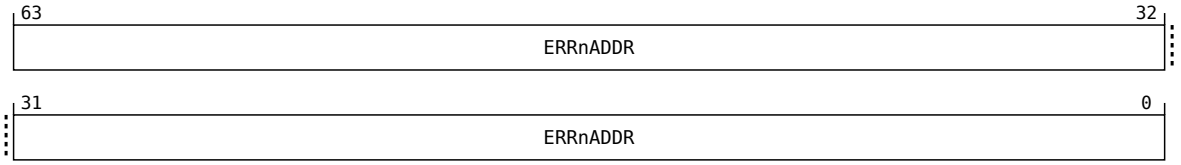
Configuration

This register is present only when FEAT_RAS is implemented. Otherwise, direct accesses to ERXADDR_EL1 are UNDEFINED.

Attributes

ERXADDR_EL1 is a 64-bit register.

Field descriptions



ERRnADDR, bits [63:0]

ERXADDR_EL1 accesses ERR<n>ADDR, where <n> is the value in ERRSELR_EL1.SEL.

Accessing ERXADDR_EL1

If ERRIDR_EL1.NUM is 0x0000 or ERRSELR_EL1.SEL is greater than or equal to ERRIDR_EL1.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXADDR_EL1 is RAZ/WI.
- Direct reads and writes of ERXADDR_EL1 are NOPs.
- Direct reads and writes of ERXADDR_EL1 are UNDEFINED.

ERR<n>ADDR describes additional constraints that also apply when ERR<n>ADDR is accessed through ERXADDR_EL1.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ERXADDR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0100	0b011

```
if !IsFeatureImplemented(FEAT_RAS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TERR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ERXADDR_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ERXADDR_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR ERXADDR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0100	0b011

```
if !IsFeatureImplemented(FEAT_RAS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TERR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        ERXADDR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    ERXADDR_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.7.5 ERXCTLR_EL1, Selected Error Record Control Register

The ERXCTLR_EL1 characteristics are:

Purpose

Accesses ERR<n>CTLR for the error record <n> selected by ERRSELR_EL1.SEL.

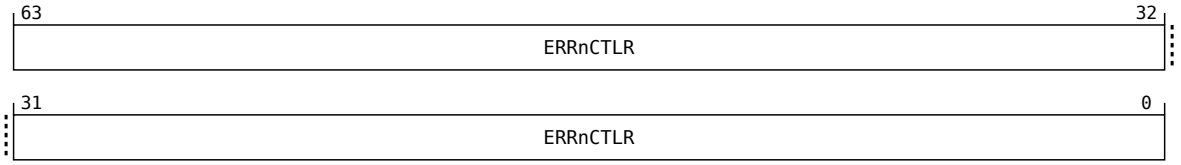
Configuration

This register is present only when FEAT_RAS is implemented. Otherwise, direct accesses to ERXCTLR_EL1 are UNDEFINED.

Attributes

ERXCTLR_EL1 is a 64-bit register.

Field descriptions



ERRnCTLR, bits [63:0]

ERXCTLR_EL1 accesses ERR<n>CTLR, where <n> is the value in ERRSELR_EL1.SEL.

Accessing ERXCTLR_EL1

If ERRIDR_EL1.NUM is 0x0000 or ERRSELR_EL1.SEL is greater than or equal to ERRIDR_EL1.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXCTLR_EL1 is RAZ/WI.
- Direct reads and writes of ERXCTLR_EL1 are NOPs.
- Direct reads and writes of ERXCTLR_EL1 are UNDEFINED.

If ERRSELR_EL1.SEL is not the index of the first error record owned by a node, then ERR<n>CTLR is not present, meaning reads and writes of ERXCTLR_EL1 are RES0.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ERXCTLR_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0100	0b001

```
if !IsFeatureImplemented(FEAT_RAS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TERR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ERXCTLR_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ERXCTLR_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR ERXCTLR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0100	0b001

```
if !IsFeatureImplemented(FEAT_RAS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TERR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        ERXCTLR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    ERXCTLR_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.7.6 ERXFR_EL1, Selected Error Record Feature Register

The ERXFR_EL1 characteristics are:

Purpose

Accesses ERR<n>FR for the error record <n> selected by ERRSELR_EL1.SEL.

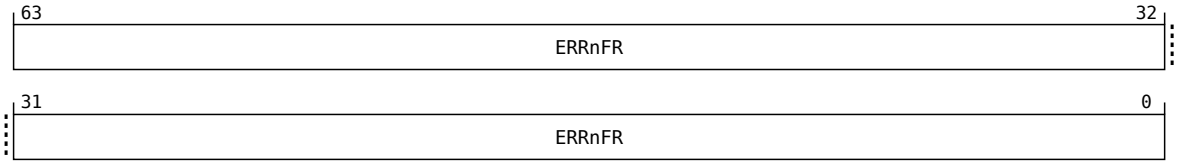
Configuration

This register is present only when FEAT_RAS is implemented. Otherwise, direct accesses to ERXFR_EL1 are UNDEFINED.

Attributes

ERXFR_EL1 is a 64-bit register.

Field descriptions



ERRnFR, bits [63:0]

ERXFR_EL1 accesses ERR<n>FR, where <n> is the value in ERRSELR_EL1.SEL.

Accessing ERXFR_EL1

If ERRIDR_EL1.NUM is `0x0000` or ERRSELR_EL1.SEL is greater than or equal to ERRIDR_EL1.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXFR_EL1 is RAZ.
- Direct reads of ERXFR_EL1 are NOPs.
- Direct reads of ERXFR_EL1 are UNDEFINED.

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, ERXFR_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0100	0b000

```
if !IsFeatureImplemented(FEAT_RAS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TERR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ERXFR_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ERXFR_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.7.7 ERXMISC0_EL1, Selected Error Record Miscellaneous Register 0

The ERXMISC0_EL1 characteristics are:

Purpose

Accesses ERR<n>MISC0 for the error record <n> selected by ERRSELR_EL1.SEL.

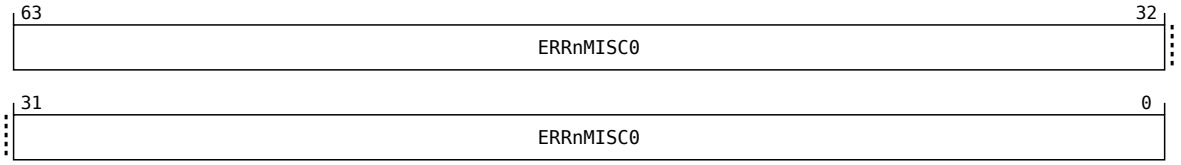
Configuration

This register is present only when FEAT_RAS is implemented. Otherwise, direct accesses to ERXMISC0_EL1 are UNDEFINED.

Attributes

ERXMISC0_EL1 is a 64-bit register.

Field descriptions



ERRnMISC0, bits [63:0]

ERXMISC0_EL1 accesses ERR<n>MISC0, where <n> is the value in ERRSELR_EL1.SEL.

Accessing ERXMISC0_EL1

If ERRIDR_EL1.NUM is `0x0000` or ERRSELR_EL1.SEL is greater than or equal to ERRIDR_EL1.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXMISC0_EL1 is RAZ/WI.
- Direct reads and writes of ERXMISC0_EL1 are NOPs.
- Direct reads and writes of ERXMISC0_EL1 are UNDEFINED.

ERR<n>MISC0 describes additional constraints that also apply when ERR<n>MISC0 is accessed through ERXMISC0_EL1.

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, ERXMISC0_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0101	0b000

```
if !IsFeatureImplemented(FEAT_RAS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TERR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ERXMISC0_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ERXMISC0_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR ERXMISC0_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0101	0b000

```
if !IsFeatureImplemented(FEAT_RAS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TERR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        ERXMISC0_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    ERXMISC0_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.7.8 ERXMISC1_EL1, Selected Error Record Miscellaneous Register 1

The ERXMISC1_EL1 characteristics are:

Purpose

Accesses ERR<n>MISC1 for the error record <n> selected by ERRSELR_EL1.SEL.

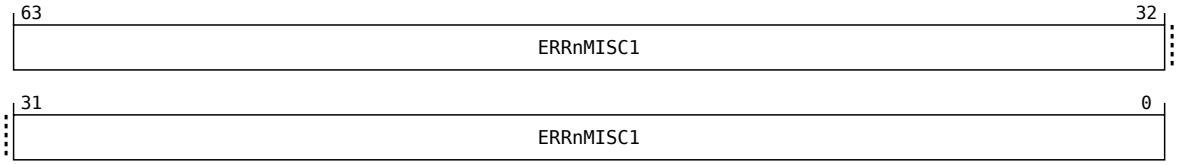
Configuration

This register is present only when FEAT_RAS is implemented. Otherwise, direct accesses to ERXMISC1_EL1 are UNDEFINED.

Attributes

ERXMISC1_EL1 is a 64-bit register.

Field descriptions



ERRnMISC1, bits [63:0]

ERXMISC1_EL1 accesses ERR<n>MISC1, where <n> is the value in ERRSELR_EL1.SEL.

Accessing ERXMISC1_EL1

If ERRIDR_EL1.NUM is `0x0000` or ERRSELR_EL1.SEL is greater than or equal to ERRIDR_EL1.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXMISC1_EL1 is RAZ/WI.
- Direct reads and writes of ERXMISC1_EL1 are NOPs.
- Direct reads and writes of ERXMISC1_EL1 are UNDEFINED.

ERR<n>MISC1 describes additional constraints that also apply when ERR<n>MISC1 is accessed through ERXMISC1_EL1.

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, ERXMISC1_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0101	0b001

```
if !IsFeatureImplemented(FEAT_RAS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TERR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ERXMISC1_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ERXMISC1_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR ERXMISC1_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0101	0b001

```
if !IsFeatureImplemented(FEAT_RAS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TERR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        ERXMISC1_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    ERXMISC1_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.7.9 ERXMISC2_EL1, Selected Error Record Miscellaneous Register 2

The ERXMISC2_EL1 characteristics are:

Purpose

Accesses ERR<n>MISC2 for the error record <n> selected by ERRSELR_EL1.SEL.

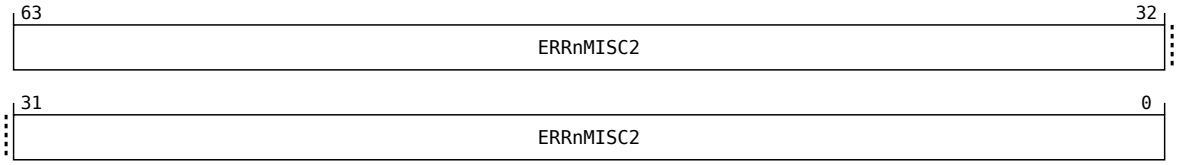
Configuration

This register is present only when FEAT_RASv1p1 is implemented. Otherwise, direct accesses to ERXMISC2_EL1 are UNDEFINED.

Attributes

ERXMISC2_EL1 is a 64-bit register.

Field descriptions



ERRnMISC2, bits [63:0]

ERXMISC2_EL1 accesses ERR<n>MISC2, where <n> is the value in ERRSELR_EL1.SEL.

Accessing ERXMISC2_EL1

If ERRIDR_EL1.NUM is 0x0000 or ERRSELR_EL1.SEL is greater than or equal to ERRIDR_EL1.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXMISC2_EL1 is RAZ/WI.
- Direct reads and writes of ERXMISC2_EL1 are NOPs.
- Direct reads and writes of ERXMISC2_EL1 are UNDEFINED.

ERR<n>MISC2 describes additional constraints that also apply when ERR<n>MISC2 is accessed through ERXMISC2_EL1.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ERXMISC2_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0101	0b010

```
if !IsFeatureImplemented(FEAT_RASv1p1) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TERR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ERXMISC2_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ERXMISC2_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR ERXMISC2_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0101	0b010

```
if !IsFeatureImplemented(FEAT_RASv1p1) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TERR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        ERXMISC2_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    ERXMISC2_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.7.10 ERXMISC3_EL1, Selected Error Record Miscellaneous Register 3

The ERXMISC3_EL1 characteristics are:

Purpose

Accesses ERR<n>MISC3 for the error record <n> selected by ERRSELR_EL1.SEL.

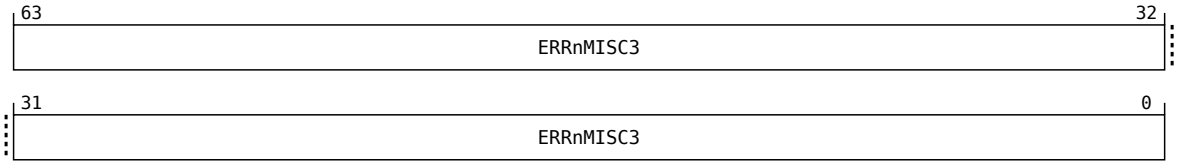
Configuration

This register is present only when FEAT_RASv1p1 is implemented. Otherwise, direct accesses to ERXMISC3_EL1 are UNDEFINED.

Attributes

ERXMISC3_EL1 is a 64-bit register.

Field descriptions



ERRnMISC3, bits [63:0]

ERXMISC3_EL1 accesses ERR<n>MISC3, where <n> is the value in ERRSELR_EL1.SEL.

Accessing ERXMISC3_EL1

If ERRIDR_EL1.NUM is 0x0000 or ERRSELR_EL1.SEL is greater than or equal to ERRIDR_EL1.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXMISC3_EL1 is RAZ/WI.
- Direct reads and writes of ERXMISC3_EL1 are NOPs.
- Direct reads and writes of ERXMISC3_EL1 are UNDEFINED.

ERR<n>MISC3 describes additional constraints that also apply when ERR<n>MISC3 is accessed through ERXMISC3_EL1.

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, ERXMISC3_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0101	0b011

```
if !IsFeatureImplemented(FEAT_RASv1p1) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TERR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ERXMISC3_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ERXMISC3_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR ERXMISC3_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0101	0b011

```
if !IsFeatureImplemented(FEAT_RASv1p1) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TERR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        ERXMISC3_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    ERXMISC3_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.7.11 ERXPFGCDN_EL1, Selected Pseudo-fault Generation Countdown Register

The ERXPFGCDN_EL1 characteristics are:

Purpose

Accesses ERR<n>PFGCDN for the error record <n> selected by ERRSELR_EL1.SEL.

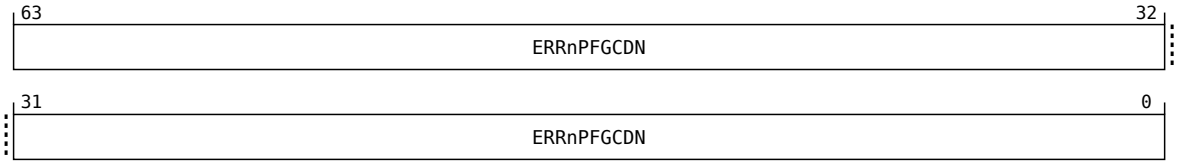
Configuration

This register is present only when FEAT_RASv1p1 is implemented. Otherwise, direct accesses to ERXPFGCDN_EL1 are UNDEFINED.

Attributes

ERXPFGCDN_EL1 is a 64-bit register.

Field descriptions



ERRnPFGCDN, bits [63:0]

ERXPFGCDN_EL1 accesses ERR<n>PFGCDN, where <n> is the value in ERRSELR_EL1.SEL.

Accessing ERXPFGCDN_EL1

If ERRIDR_EL1.NUM is `0x0000` or ERRSELR_EL1.SEL is greater than or equal to ERRIDR_EL1.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXPFGCDN_EL1 is RAZ/WI.
- Direct reads and writes of ERXPFGCDN_EL1 are NOPs.
- Direct reads and writes of ERXPFGCDN_EL1 are UNDEFINED.

If ERRSELR_EL1.SEL selects an error record owned by a node that does not implement the Common Fault Injection Model Extension, then one of the following occurs:

- ERXPFGCDN_EL1 is RAZ/WI.
- Direct reads and writes of ERXPFGCDN_EL1 are NOPs.
- Direct reads and writes of ERXPFGCDN_EL1 are UNDEFINED.

Note

A node does not implement the Common Fault Injection Model Extension if ERR<q>FR.INJ reads as `0b00`. <q> is the index of the first error record owned by the same node as error record <n>, where <n> is the value in ERRSELR_EL1.SEL. If the node owns a single record then q = n.

If ERRSELR_EL1.SEL is not the index of the first error record owned by a node, then ERR<n>PFGCDN is not present, meaning reads and writes of ERXPFGCDN_EL1 are RES0.

ERR<n>PFGCDN describes additional constraints that also apply when ERR<n>PFGCDN is accessed through ERXPFGCDN_EL1.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MRS <Xt>, ERXPFGCDN_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0100	0b110

```
if !IsFeatureImplemented(FEAT_RASv1p1) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.FIEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ERXPFGCDN_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ERXPFGCDN_EL1;
```

MSR ERXPFGCDN_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0100	0b110

```
if !IsFeatureImplemented(FEAT_RASv1p1) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.FIEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        ERXPFGCDN_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    ERXPFGCDN_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.7.12 ERXPFGCTL_EL1, Selected Pseudo-fault Generation Control Register

The ERXPFGCTL_EL1 characteristics are:

Purpose

Accesses ERR<n>PFGCTL for the error record <n> selected by ERRSELR_EL1.SEL.

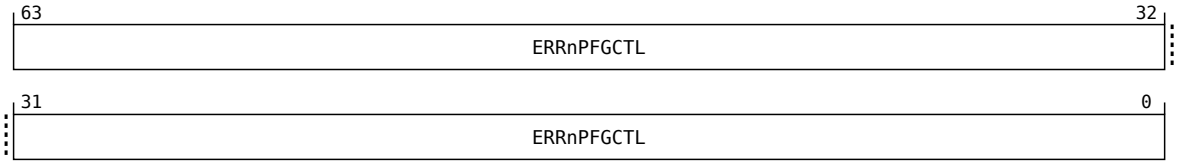
Configuration

This register is present only when FEAT_RASv1p1 is implemented. Otherwise, direct accesses to ERXPFGCTL_EL1 are UNDEFINED.

Attributes

ERXPFGCTL_EL1 is a 64-bit register.

Field descriptions



ERRnPFGCTL, bits [63:0]

ERXPFGCTL_EL1 accesses ERR<n>PFGCTL, where <n> is the value in ERRSELR_EL1.SEL.

Accessing ERXPFGCTL_EL1

If ERRIDR_EL1.NUM is `0x0000` or ERRSELR_EL1.SEL is greater than or equal to ERRIDR_EL1.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXPFGCTL_EL1 is RAZ/WI.
- Direct reads and writes of ERXPFGCTL_EL1 are NOPs.
- Direct reads and writes of ERXPFGCTL_EL1 are UNDEFINED.

If ERRSELR_EL1.SEL selects an error record owned by a node that does not implement the Common Fault Injection Model Extension, then one of the following occurs:

- ERXPFGCTL_EL1 is RAZ/WI.
- Direct reads and writes of ERXPFGCTL_EL1 are NOPs.
- Direct reads and writes of ERXPFGCTL_EL1 are UNDEFINED.

Note

A node does not implement the Common Fault Injection Model Extension if ERR<q>FR.INJ reads as `0b00`. <q> is the index of the first error record owned by the same node as error record <n>, where <n> is the value in ERRSELR_EL1.SEL. If the node owns a single record then q = n.

If ERRSELR_EL1.SEL is not the index of the first error record owned by a node, then ERR<n>PFGCTL is not present, meaning reads and writes of ERXPFGCTL_EL1 are RES0.

ERR<n>PFGCTL describes additional constraints that also apply when ERR<n>PFGCTL is accessed through ERXPFGCTL_EL1.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MRS <Xt>, ERXPFGCTL_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0100	0b101

```
if !IsFeatureImplemented(FEAT_RASv1p1) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.FIEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ERXPFGCTL_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ERXPFGCTL_EL1;
```

MSR ERXPFGCTL_EL1, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0100	0b101

```
if !IsFeatureImplemented(FEAT_RASv1p1) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.FIEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        ERXPFGCTL_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    ERXPFGCTL_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.7.13 ERXPFGF_EL1, Selected Pseudo-fault Generation Feature Register

The ERXPFGF_EL1 characteristics are:

Purpose

Accesses ERR<n>PFGF for the error record <n> selected by ERRSELR_EL1.SEL.

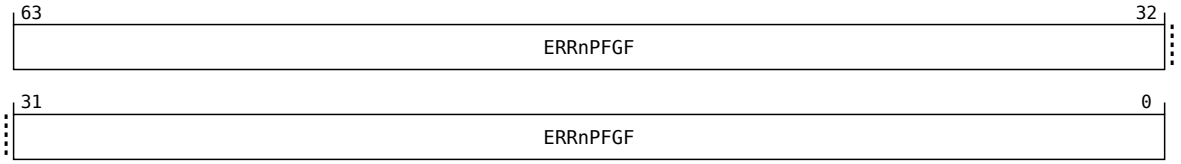
Configuration

This register is present only when FEAT_RASv1p1 is implemented. Otherwise, direct accesses to ERXPFGF_EL1 are UNDEFINED.

Attributes

ERXPFGF_EL1 is a 64-bit register.

Field descriptions



ERRnPFGF, bits [63:0]

ERXPFGF_EL1 accesses ERR<n>PFGF, where <n> is the value in ERRSELR_EL1.SEL.

Accessing ERXPFGF_EL1

If ERRIDR_EL1.NUM is `0x0000` or ERRSELR_EL1.SEL is greater than or equal to ERRIDR_EL1.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXPFGF_EL1 is RAZ.
- Direct reads of ERXPFGF_EL1 are NOPs.
- Direct reads of ERXPFGF_EL1 are UNDEFINED.

If ERRSELR_EL1.SEL selects an error record owned by a node that does not implement the Common Fault Injection Model Extension, then one of the following occurs:

- ERXPFGF_EL1 is RAZ.
- Direct reads of ERXPFGF_EL1 are NOPs.
- Direct reads of ERXPFGF_EL1 are UNDEFINED.

Note

A node does not implement the Common Fault Injection Model Extension if ERR<q>FR.INJ reads as `0b00`. <q> is the index of the first error record owned by the same node as error record <n>, where <n> is the value in ERRSELR_EL1.SEL. If the node owns a single record then q = n.

If ERRSELR_EL1.SEL is not the index of the first error record owned by a node, then ERR<n>PFGF is not present, meaning reads of ERXPFGF_EL1 are RES0.

ERR<n>PFGF describes additional constraints that also apply when ERR<n>PFGF is accessed through ERXPFGF_EL1.

Accesses to this register use the following encodings in the System register encoding space:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MRS <Xt>, ERXPFGF_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0100	0b100

```
if !IsFeatureImplemented(FEAT_RASv1p1) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.FIEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ERXPFGF_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ERXPFGF_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.7.14 ERXSTATUS_EL1, Selected Error Record Primary Status Register

The ERXSTATUS_EL1 characteristics are:

Purpose

Accesses ERR<n>STATUS for the error record <n> selected by ERRSELR_EL1.SEL.

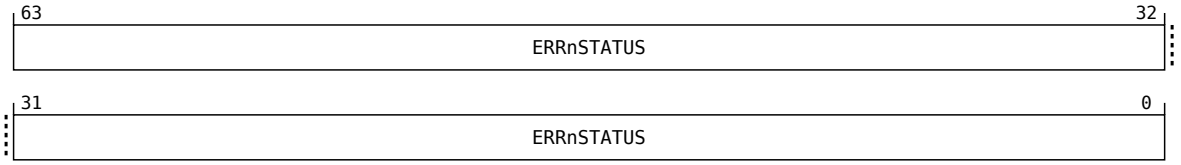
Configuration

This register is present only when FEAT_RAS is implemented. Otherwise, direct accesses to ERXSTATUS_EL1 are UNDEFINED.

Attributes

ERXSTATUS_EL1 is a 64-bit register.

Field descriptions



ERRnSTATUS, bits [63:0]

ERXSTATUS_EL1 accesses ERR<n>STATUS, where <n> is the value in ERRSELR_EL1.SEL.

Accessing ERXSTATUS_EL1

If ERRIDR_EL1.NUM is `0x0000` or ERRSELR_EL1.SEL is greater than or equal to ERRIDR_EL1.NUM, then one of the following occurs:

- An UNKNOWN error record is selected.
- ERXSTATUS_EL1 is RAZ/WI.
- Direct reads and writes of ERXSTATUS_EL1 are NOPs.
- Direct reads and writes of ERXSTATUS_EL1 are UNDEFINED.

ERR<n>STATUS describes additional constraints that also apply when ERR<n>STATUS is accessed through ERXSTATUS_EL1.

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, ERXSTATUS_EL1`

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0100	0b010

```
if !IsFeatureImplemented(FEAT_RAS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TERR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = ERXSTATUS_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = ERXSTATUS_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR ERXSTATUS_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0100	0b010

```
if !IsFeatureImplemented(FEAT_RAS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.TERR == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        ERXSTATUS_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    ERXSTATUS_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.7.15 VDISR_EL2, Virtual Deferred Interrupt Status Register (EL2)

The VDISR_EL2 characteristics are:

Purpose

Records that a virtual SError exception has been consumed by an ESB instruction executed at EL1 or EL0.

An indirect write to VDISR_EL2 made by an ESB instruction does not require an explicit synchronization operation for the value written to be observed by a direct read of one of the following registers occurring in program order after the ESB instruction:

- DISR_EL1.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

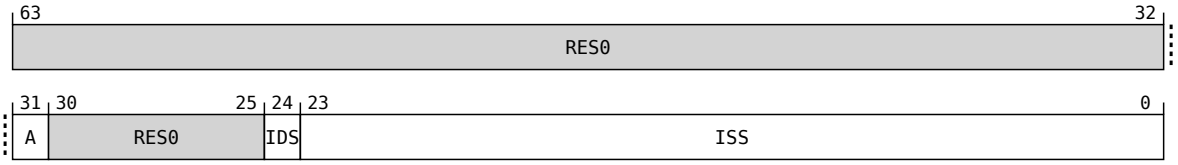
This register has no effect if EL2 is not enabled in the current Security state.

This register is present only when FEAT_RAS is implemented. Otherwise, direct accesses to VDISR_EL2 are UNDEFINED.

Attributes

VDISR_EL2 is a 64-bit register.

Field descriptions



Bits [63:32]

Reserved, RES0.

A, bit [31]

Set to 1 when an ESB instruction defers a virtual SError exception.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [30:25]

Reserved, RES0.

IDS, bit [24]

The value copied from VSESR_EL2.IDS.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ISS, bits [23:0]

The value copied from VSESR_EL2.ISS.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing VDISR_EL2

An indirect write to VDISR_EL2 made by an ESB instruction does not require an explicit synchronization operation for the value that is written to be observed by a direct read of one of the following registers occurring in program order after the ESB instruction:

- DISR_EL1.

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, VDISR_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b1100	0b0001	0b001

```
if !IsFeatureImplemented(FEAT_RAS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = VDISR_EL2;
```

MSR VDISR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1100	0b0001	0b001

```
if !IsFeatureImplemented(FEAT_RAS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    VDISR_EL2 = X[t, 64];
```

MRS <Xt>, DISR_EL1

op0	op1	CRn	CRm	op2
0b11	0b000	0b1100	0b0001	0b001

```
if !IsFeatureImplemented(FEAT_RAS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.AMO == '1' then
        X[t, 64] = VDISR_EL2;
    else
        X[t, 64] = DISR_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = DISR_EL1;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR DISR_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1100	0b0001	0b001

```
if !IsFeatureImplemented(FEAT_RAS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if HCR_EL2.AMO == '1' then
        VDISR_EL2 = X[t, 64];
    else
        DISR_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    DISR_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.7.16 VSESR_EL2, Virtual SError Exception Syndrome Register (EL2)

The VSESR_EL2 characteristics are:

Purpose

Provides the syndrome value reported to software on taking a virtual SError exception to EL1, or on executing an ESB instruction at EL1.

When the virtual SError exception injected using HCR_EL2.VSE is taken to EL1 using AArch64, then the syndrome value is reported in ESR_EL1.

When the virtual SError exception injected using HCR_EL2.VSE is deferred by an ESB instruction, then the syndrome value is written to VDISR_EL2.

Configuration

If EL2 is not implemented, this register is RES0 from EL3.

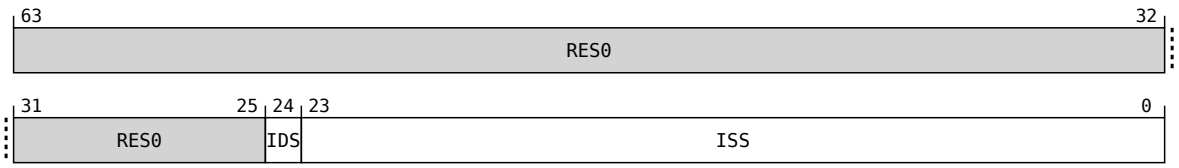
This register has no effect if EL2 is not enabled in the current Security state.

This register is present only when FEAT_RAS is implemented. Otherwise, direct accesses to VSESR_EL2 are UNDEFINED.

Attributes

VSESR_EL2 is a 64-bit register.

Field descriptions



Bits [63:25]

Reserved, RES0.

IDS, bit [24]

When a virtual SError exception is taken to EL1 using AArch64, ESR_EL1[24] is set to VSESR_EL2.IDS.

When a virtual SError exception is deferred by an ESB instruction, VDISR_EL2[24] is set to VSESR_EL2.IDS.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ISS, bits [23:0]

When a virtual SError exception is taken to EL1 using AArch64, ESR_EL1[23:0] is set to VSESR_EL2.ISS.

When a virtual SError exception is deferred by an ESB instruction, VDISR_EL2[23:0] is set to VSESR_EL2.ISS.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing VSESR_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, VSESR_EL2
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b100	0b0101	0b0010	0b011

```
if !IsFeatureImplemented(FEAT_RAS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = VSESR_EL2;
```

MSR VSESR_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b0101	0b0010	0b011

```
if !IsFeatureImplemented(FEAT_RAS) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    VSESR_EL2 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.8 Generic Timer registers

This section lists the Generic Timer registers in AArch64.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.8.1 CNTFRQ_EL0, Counter-timer Frequency Register

The CNTFRQ_EL0 characteristics are:

Purpose

This register is provided so that software can discover the effective frequency of the system counter. It must be programmed with this value as part of system initialization. The value of the register is not interpreted by hardware.

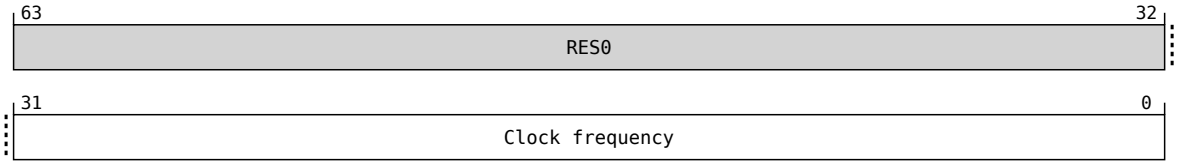
Configuration

There are no configuration notes.

Attributes

CNTFRQ_EL0 is a 64-bit register.

Field descriptions



Bits [63:32]

Reserved, RES0.

ClockFreq, bits [31:0]

Clock frequency. Indicates the effective frequency of the system counter, in Hz.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTFRQ_EL0

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, CNTFRQ_EL0`

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0000	0b000

```
if PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1.<EL0PCTEN,EL0VCTEN> == '00' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        X[t, 64] = CNTFRQ_EL0;
elsif PSTATE.EL == EL1 then
    X[t, 64] = CNTFRQ_EL0;
elsif PSTATE.EL == EL2 then
    X[t, 64] = CNTFRQ_EL0;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR CNTFRQ_EL0, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0000	0b000

```
if IsHighestEL(PSTATE.EL) then
    CNTFRQ_EL0 = X[t, 64];
else
    UNDEFINED;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.8.2 CNTHCTL_EL2, Counter-timer Hypervisor Control Register

The CNTHCTL_EL2 characteristics are:

Purpose

Controls the generation of an event stream from the physical counter, and access from EL1 to the physical counter and the EL1 physical timer.

Configuration

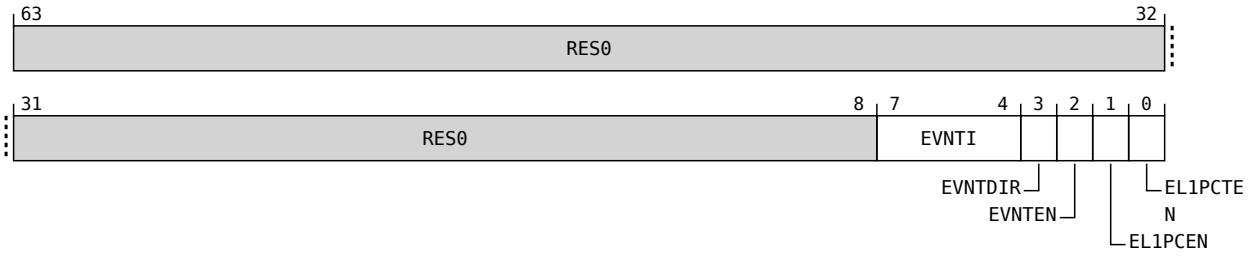
There are no configuration notes.

Attributes

CNTHCTL_EL2 is a 64-bit register.

Field descriptions

!ELIsInHost(EL2)



The following field descriptions apply in all Armv8.0 implementations.

Bits [63:8]

Reserved, RES0.

EVNTI, bits [7:4]

Selects which bit of CNTPCT_EL0, as seen from EL2,is the trigger for the event stream generated from that counter when that stream is enabled.

This field selects a trigger bit in the range 0 to 15 of CNTPCT_EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EVNTDIR, bit [3]

Controls which transition of the CNTPCT_EL0 trigger bit, as seen from EL2 and defined by EVNTI, generates an event when the event stream is enabled.

EVNTDIR	Meaning
0b0	A 0 to 1 transition of the trigger bit triggers an event.
0b1	A 1 to 0 transition of the trigger bit triggers an event.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EVNTEN, bit [2]

Enables the generation of an event stream from CNTPCT_EL0 as seen from EL2.

EVNTEN	Meaning
0b0	Disables the event stream.
0b1	Enables the event stream.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EL1PCEN, bit [1]

Traps EL0 and EL1 accesses to the EL1 physical timer registers to EL2 when EL2 is enabled in the current Security state, as follows:

- In AArch64 state, accesses to CNTP_CTL_EL0, CNTP_CVAL_EL0, CNTP_TVAL_EL0 are trapped to EL2, reported using EC syndrome value 0x18, unless they are trapped by CNTKCTL_EL1.EL0PTEN.

EL1PCEN	Meaning
0b0	EL0 and EL1 accesses to the specified registers are trapped to EL2.
0b1	This control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EL1PCTEN, bit [0]

Traps EL0 and EL1 accesses to the EL1 physical counter registers to EL2 when EL2 is enabled in the current Security state, as follows:

- In AArch64 state, accesses to CNTPCT_EL0 and CNTPCTSS_EL0 are trapped to EL2 and reported using EC syndrome value 0x18, unless they are trapped by CNTKCTL_EL1.EL0PCTEN.

EL1PCTEN	Meaning
0b0	EL0 and EL1 accesses to the specified registers are trapped to EL2.
0b1	This control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTHCTL_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, CNTHCTL_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = CNTHCTL_EL2;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR CNTHCTL_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    CNTHCTL_EL2 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.8.3 CNTHPS_CTL_EL2, Counter-timer Secure Physical Timer Control Register (EL2)

The CNTHPS_CTL_EL2 characteristics are:

Purpose

Control register for the Secure EL2 physical timer.

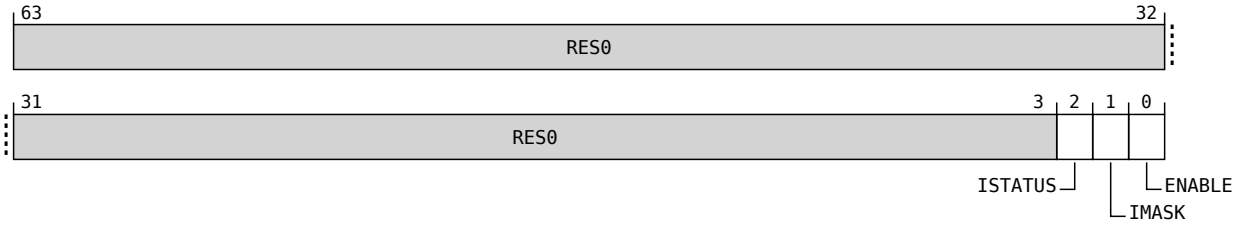
Configuration

This register is present only when FEAT_SEL2 is implemented. Otherwise, direct accesses to CNTHPS_CTL_EL2 are UNDEFINED.

Attributes

CNTHPS_CTL_EL2 is a 64-bit register.

Field descriptions



Bits [63:3]

Reserved, RES0.

ISTATUS, bit [2]

The status of the timer. This bit indicates whether the timer condition is met:

ISTATUS	Meaning
0b0	Timer condition is not met.
0b1	Timer condition is met.

When the value of the CNTHPS_CTL_EL2.ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer interrupt is asserted.

When the value of the CNTHPS_CTL_EL2.ENABLE bit is 0, the ISTATUS field is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RO.

IMASK, bit [1]

Timer interrupt mask bit. Permitted values are:

IMASK	Meaning
0b0	Timer interrupt is not masked by the IMASK bit.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IMASK	Meaning
0b1	Timer interrupt is masked by the IMASK bit.

For more information, see the description of the ISTATUS bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ENABLE, bit [0]

Enables the timer. Permitted values are:

ENABLE	Meaning
0b0	Timer disabled.
0b1	Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTHPS_TVAL_EL2 continues to count down.

Note

Disabling the output signal might be a power-saving option.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTHPS_CTL_EL2

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, CNTHPS_CTL_EL2

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0101	0b001

```
if !IsFeatureImplemented(FEAT_SEL2) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = CNTHPS_CTL_EL2;
```

MSR CNTHPS_CTL_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0101	0b001

```
if !IsFeatureImplemented(FEAT_SEL2) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    CNTHPS_CTL_EL2 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.8.4 CNTHPS_CVAL_EL2, Counter-timer Secure Physical Timer CompareValue Register (EL2)

The CNTHPS_CVAL_EL2 characteristics are:

Purpose

Holds the compare value for the Secure EL2 physical timer.

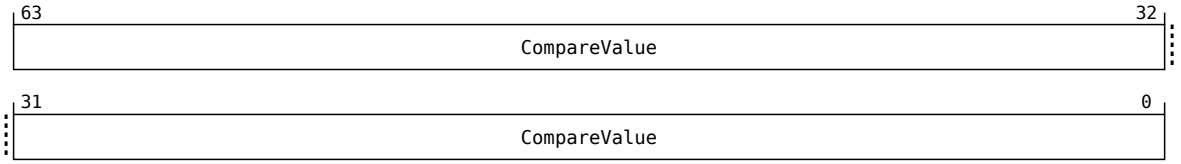
Configuration

This register is present only when FEAT_SEL2 is implemented. Otherwise, direct accesses to CNTHPS_CVAL_EL2 are UNDEFINED.

Attributes

CNTHPS_CVAL_EL2 is a 64-bit register.

Field descriptions



CompareValue, bits [63:0]

Holds the EL2 physical timer CompareValue.

When CNTHPS_CTL_EL2.ENABLE is 1, and TimerConditionMet is TRUE for the Secure EL2 physical timer, the timer condition is met and all of the following are true:

- CNTHPS_CTL_EL2.ISTATUS is set to 1.
- If CNTHPS_CTL_EL2.IMASK is 0, an interrupt is generated.

TimerConditionMet is defined by 'Operation of the CompareValue views of the timers'.

The CompareValue view of the timer acts like a 64-bit upcounter timer.

When CNTHPS_CTL_EL2.ENABLE is 0, the timer condition is not met, but CNTPCT_EL0 continues to count.

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTHPS_CVAL_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, CNTHPS_CVAL_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0101	0b010

```
if !IsFeatureImplemented(FEAT_SEL2) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = CNTHPS_CVAL_EL2;
```

MSR CNTHPS_CVAL_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0101	0b010

```
if !IsFeatureImplemented(FEAT_SEL2) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    CNTHPS_CVAL_EL2 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.8.5 CNTHPS_TVAL_EL2, Counter-timer Secure Physical Timer TimerValue Register (EL2)

The CNTHPS_TVAL_EL2 characteristics are:

Purpose

Holds the timer value for the Secure EL2 physical timer.

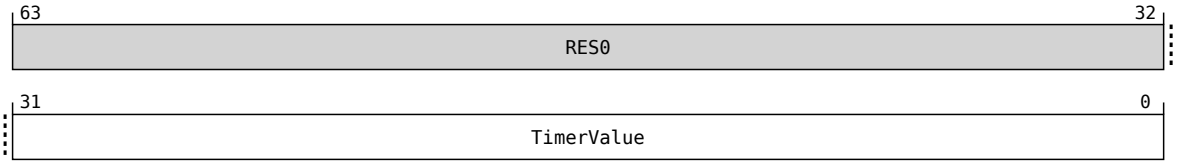
Configuration

This register is present only when FEAT_SEL2 is implemented. Otherwise, direct accesses to CNTHPS_TVAL_EL2 are UNDEFINED.

Attributes

CNTHPS_TVAL_EL2 is a 64-bit register.

Field descriptions



Bits [63:32]

Reserved, RES0.

TimerValue, bits [31:0]

The TimerValue view of the EL2 physical timer.

On a read of this register:

- If CNTHPS_CTL_EL2.ENABLE is 0, the value returned is UNKNOWN.
- If CNTHPS_CTL_EL2.ENABLE is 1, the value returned is (CNTHPS_CVAL_EL2 - CNTPCT_EL0).

On a write of this register, CNTHPS_CVAL_EL2 is set to (CNTPCT_EL0 + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When CNTHPS_CTL_EL2.ENABLE is 1, the timer condition is met when (CNTPCT_EL0 - CNTHPS_CVAL_EL2) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:

- CNTHPS_CTL_EL2.ISTATUS is set to 1.
- If CNTHPS_CTL_EL2.IMASK is 0, an interrupt is generated.

When CNTHPS_CTL_EL2.ENABLE is 0, the TimerValue cannot be read but continues to decrement. When the timer is enabled, the TimerValue represents the elapsed time whether that time was spent enabled or disabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTHPS_TVAL_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, CNTHPS_TVAL_EL2
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0101	0b000

```
if !IsFeatureImplemented(FEAT_SEL2) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if CNTHPS_CTL_EL2.ENABLE == '0' then
        X[t, 64] = bits(64) UNKNOWN;
    else
        X[t, 64] = ZeroExtend((CNTHPS_CVAL_EL2 - PhysicalCountInt())<31:0>, 64);
```

MSR CNTHPS_TVAL_EL2, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0101	0b000

```
if !IsFeatureImplemented(FEAT_SEL2) then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    CNTHPS_CVAL_EL2 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.8.6 CNTKCTL_EL1, Counter-timer Kernel Control Register

The CNTKCTL_EL1 characteristics are:

Purpose

This register controls the generation of an event stream from the virtual counter, and access from EL0 to the physical counter, virtual counter, EL1 physical timers, and the virtual timer.

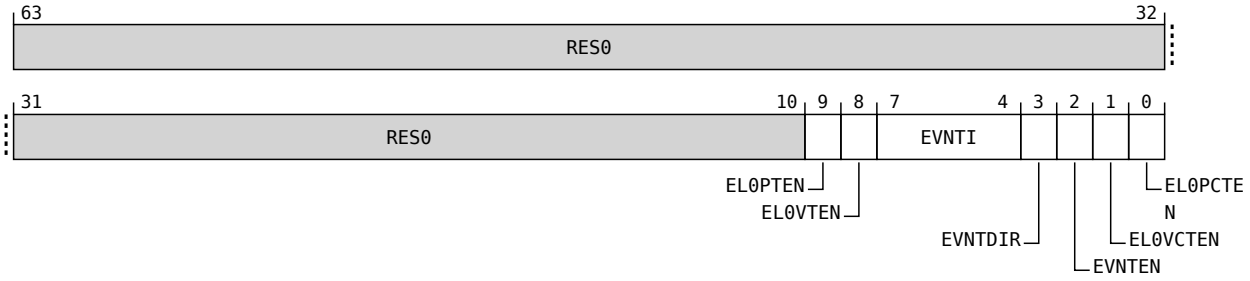
Configuration

There are no configuration notes.

Attributes

CNTKCTL_EL1 is a 64-bit register.

Field descriptions



Bits [63:10]

Reserved, RES0.

EL0PTEN, bit [9]

Traps EL0 accesses to the physical timer registers to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, as follows:

- In AArch64 state, the following registers are trapped and reported using EC syndrome value `0x18`:
  - CNTP_CTL_EL0, CNTP_CVAL_EL0, and CNTP_TVAL_EL0.

EL0PTEN	Meaning
`0b0`	EL0 accesses to the physical timer registers are trapped to EL1.
`0b1`	This control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EL0VTEN, bit [8]

Traps EL0 accesses to the virtual timer registers to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, as follows:

- In AArch64 state, accesses to the following registers are trapped and reported using EC syndrome value `0x18`:
  - CNTV_CTL_EL0, CNTV_CVAL_EL0, and CNTV_TVAL_EL0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EL0VTEN	Meaning
0b0	EL0 accesses to the virtual timer registers are trapped.
0b1	This control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EVNTI, bits [7:4]

Selects which bit of CNTVCT_EL0, as seen from EL1, is the trigger for the event stream generated from that counter when that stream is enabled.

This field selects a trigger bit in the range 0 to 15 of CNTVCT_EL0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EVNTDIR, bit [3]

Controls which transition of the CNTVCT_EL0 trigger bit, as seen from EL1 and defined by EVNTI, generates an event when the event stream is enabled.

EVNTDIR	Meaning
0b0	A 0 to 1 transition of the trigger bit triggers an event.
0b1	A 1 to 0 transition of the trigger bit triggers an event.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EVNTEN, bit [2]

Enables the generation of an event stream from CNTVCT_EL0 as seen from EL1.

EVNTEN	Meaning
0b0	Disables the event stream.
0b1	Enables the event stream.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EL0VCTEN, bit [1]

Traps EL0 accesses to the frequency register and virtual counter registers to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, as follows:

- In AArch64 state, accesses to the following registers are trapped and reported using EC syndrome value 0x18:
  - CNTVCT_EL0.
  - CNTVCTSS_EL0.
  - If CNTKCTL_EL1.EL0PCTEN is 0, CNTFRQ_EL0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EL0VCTEN	Meaning
0b0	EL0 accesses to the frequency register and virtual counter registers are trapped.
0b1	This control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

EL0PCTEN, bit [0]

Traps EL0 accesses to the frequency register and physical counter register to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, as follows:

- In AArch64 state, the following registers are trapped and reported using EC syndrome value 0x18:
  - CNTPCT_EL0
  - CNTPCTSS_EL0
  - If CNTKCTL_EL1.EL0VCTEN is 0, CNTFRQ_EL0.

EL0PCTEN	Meaning
0b0	EL0 accesses to the frequency register and physical counter register are trapped.
0b1	This control does not cause any instructions to be trapped.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTKCTL_EL1

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, CNTKCTL_EL1
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1110	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    X[t, 64] = CNTKCTL_EL1;
elsif PSTATE.EL == EL2 then
    X[t, 64] = CNTKCTL_EL1;
```

```
MSR CNTKCTL_EL1, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b000	0b1110	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    CNTKCTL_EL1 = X[t, 64];
elsif PSTATE.EL == EL2 then
    CNTKCTL_EL1 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.8.7 CNTP_CTL_EL0, Counter-timer Physical Timer Control Register

The CNTP_CTL_EL0 characteristics are:

Purpose

Control register for the EL1 physical timer.

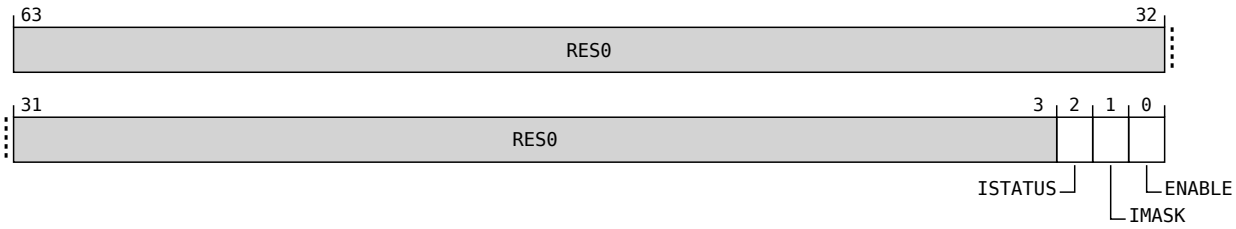
Configuration

There are no configuration notes.

Attributes

CNTP_CTL_EL0 is a 64-bit register.

Field descriptions



Bits [63:3]

Reserved, RES0.

ISTATUS, bit [2]

The status of the timer. This bit indicates whether the timer condition is met:

ISTATUS	Meaning
0b0	Timer condition is not met.
0b1	Timer condition is met.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer interrupt is asserted.

When the value of the ENABLE bit is 0, the ISTATUS field is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RO.

IMASK, bit [1]

Timer interrupt mask bit. Permitted values are:

IMASK	Meaning
0b0	Timer interrupt is not masked by the IMASK bit.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b1	Timer interrupt is masked by the IMASK bit.

For more information, see the description of the ISTATUS bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ENABLE, bit [0]

Enables the timer. Permitted values are:

ENABLE	Meaning
0b0	Timer disabled.
0b1	Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTP_TVAL_EL0 continues to count down.

Note

Disabling the output signal might be a power-saving option.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTP_CTL_EL0

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, CNTP_CTL_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0010	0b001

```
if PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1.EL0PTEN == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif !ELIsInHost(EL2) && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = CNTP_CTL_EL0;
elsif PSTATE.EL == EL1 then
    if !ELIsInHost(EL2) && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL2) then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = CNTP_CTL_EL0;
elsif PSTATE.EL == EL2 then
    X[t, 64] = CNTP_CTL_EL0;
```

MSR CNTP_CTL_EL0, <Xt>

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0010	0b001

```
if PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1.EL0PTEN == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif !ELIsInHost(EL2) && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        CNTP_CTL_EL0 = X[t, 64];
elsif PSTATE.EL == EL1 then
    if !ELIsInHost(EL2) && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL2) then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        CNTP_CTL_EL0 = X[t, 64];
elsif PSTATE.EL == EL2 then
    CNTP_CTL_EL0 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.8.8 CNTP_CVAL_EL0, Counter-timer Physical Timer CompareValue Register

The CNTP_CVAL_EL0 characteristics are:

Purpose

Holds the compare value for the EL1 physical timer.

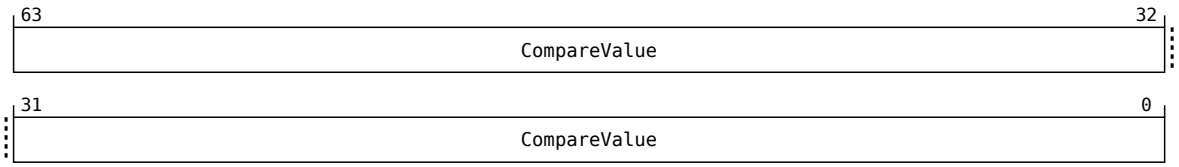
Configuration

There are no configuration notes.

Attributes

CNTP_CVAL_EL0 is a 64-bit register.

Field descriptions



CompareValue, bits [63:0]

Holds the EL1 physical timer CompareValue.

When CNTP_CTL_EL0.ENABLE is 1, and TimerConditionMet is TRUE for the EL1 physical timer, the timer condition is met and all of the following are true:

- CNTP_CTL_EL0.ISTATUS is set to 1.
- If CNTP_CTL_EL0.IMASK is 0, an interrupt is generated.

TimerConditionMet is defined by 'Operation of the CompareValue views of the timers'.

The CompareValue view of the timer acts like a 64-bit upcounter timer.

When CNTP_CTL_EL0.ENABLE is 0, the timer condition is not met, but CNTPCT_EL0 continues to count.

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTP_CVAL_EL0

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, CNTP_CVAL_EL0
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0010	0b010

```
if PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1.EL0PTEN == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif !ELIsInHost(EL2) && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = CNTP_CVAL_EL0;
elsif PSTATE.EL == EL1 then
    if !ELIsInHost(EL2) && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL2) then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = CNTP_CVAL_EL0;
elsif PSTATE.EL == EL2 then
    X[t, 64] = CNTP_CVAL_EL0;
```

MSR CNTP_CVAL_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0010	0b010

```
if PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1.EL0PTEN == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif !ELIsInHost(EL2) && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        CNTP_CVAL_EL0 = X[t, 64];
elsif PSTATE.EL == EL1 then
    if !ELIsInHost(EL2) && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL2) then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        CNTP_CVAL_EL0 = X[t, 64];
elsif PSTATE.EL == EL2 then
    CNTP_CVAL_EL0 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.8.9 CNTP_TVAL_EL0, Counter-timer Physical Timer TimerValue Register

The CNTP_TVAL_EL0 characteristics are:

Purpose

Holds the timer value for the EL1 physical timer.

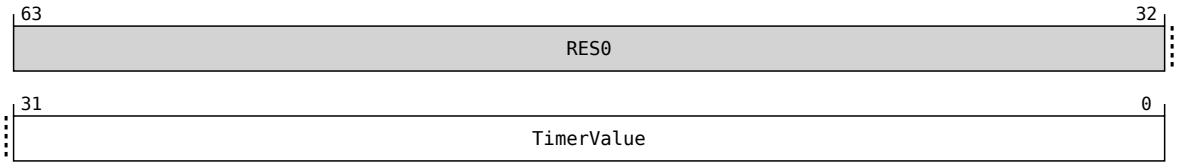
Configuration

There are no configuration notes.

Attributes

CNTP_TVAL_EL0 is a 64-bit register.

Field descriptions



Bits [63:32]

Reserved, RES0.

TimerValue, bits [31:0]

The TimerValue view of the EL1 physical timer.

On a read of this register:

- If CNTP_CTL_EL0.ENABLE is 0, the value returned is UNKNOWN.
- If CNTP_CTL_EL0.ENABLE is 1, the value returned is (CNTP_CVAL_EL0 - CNTPCT_EL0).

On a write of this register, CNTP_CVAL_EL0 is set to (CNTPCT_EL0 + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When CNTP_CTL_EL0.ENABLE is 1, the timer condition is met when (CNTPCT_EL0 - CNTP_CVAL_EL0) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:

- CNTP_CTL_EL0.ISTATUS is set to 1.
- If CNTP_CTL_EL0.IMASK is 0, an interrupt is generated.

When CNTP_CTL_EL0.ENABLE is 0, the TimerValue cannot be read but continues to decrement. When the timer is enabled, the TimerValue represents the elapsed time whether that time was spent enabled or disabled.

Note

The value of CNTPCT_EL0 used in these calculations is the value seen at the Exception level that the CNTPCT_EL0 register is being read or written from.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTP_TVAL_EL0

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, CNTP_TVAL_EL0
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0010	0b000

```
if PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1.EL0PTEN == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif !ELIsInHost(EL2) && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if CNTP_CTL_EL0.ENABLE == '0' then
            X[t, 64] = bits(64) UNKNOWN;
        else
            X[t, 64] = ZeroExtend((CNTP_CVAL_EL0 - PhysicalCountInt())<31:0>, 64);
elsif PSTATE.EL == EL1 then
    if !ELIsInHost(EL2) && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL2) then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        if CNTP_CTL_EL0.ENABLE == '0' then
            X[t, 64] = bits(64) UNKNOWN;
        else
            X[t, 64] = ZeroExtend((CNTP_CVAL_EL0 - PhysicalCountInt())<31:0>, 64);
elsif PSTATE.EL == EL2 then
    if CNTP_CTL_EL0.ENABLE == '0' then
        X[t, 64] = bits(64) UNKNOWN;
    else
        X[t, 64] = ZeroExtend((CNTP_CVAL_EL0 - PhysicalCountInt())<31:0>, 64);
```

MSR CNTP_TVAL_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0010	0b000

```
if PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1.EL0PTEN == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif !ELIsInHost(EL2) && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        CNTP_CVAL_EL0 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
elsif PSTATE.EL == EL1 then
    if !ELIsInHost(EL2) && CNTHCTL_EL2.EL1PCEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif ELIsInHost(EL2) then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        CNTP_CVAL_EL0 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
elsif PSTATE.EL == EL2 then
    CNTP_CVAL_EL0 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.8.10 CNTPCT_EL0, Counter-timer Physical Count Register

The CNTPCT_EL0 characteristics are:

Purpose

Reads of CNTPCT_EL0 return the 64-bit physical count value minus a physical offset.

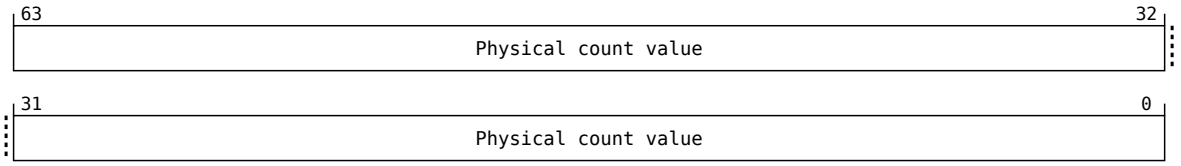
Configuration

There are no configuration notes.

Attributes

CNTPCT_EL0 is a 64-bit register.

Field descriptions



PhysicalCount, bits [63:0]

Physical count value.

Reads of CNTPCT_EL0 return PhysicalCountInt<63:0>.

PhysicalCountInt is defined by 'The Physical Counter'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTPCT_EL0

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, CNTPCT_EL0`

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0000	0b001

```
if PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1.EL0PCTEN == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif !ELIsInHost(EL2) && CNTHCTL_EL2.EL1PCTEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = PhysicalCountInt();
elsif PSTATE.EL == EL1 then
    if CNTHCTL_EL2.EL1PCTEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        X[t, 64] = PhysicalCountInt();
elsif PSTATE.EL == EL2 then
    X[t, 64] = PhysicalCountInt();
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.8.11 CNTV_CTL_EL0, Counter-timer Virtual Timer Control Register

The CNTV_CTL_EL0 characteristics are:

Purpose

Control register for the virtual timer.

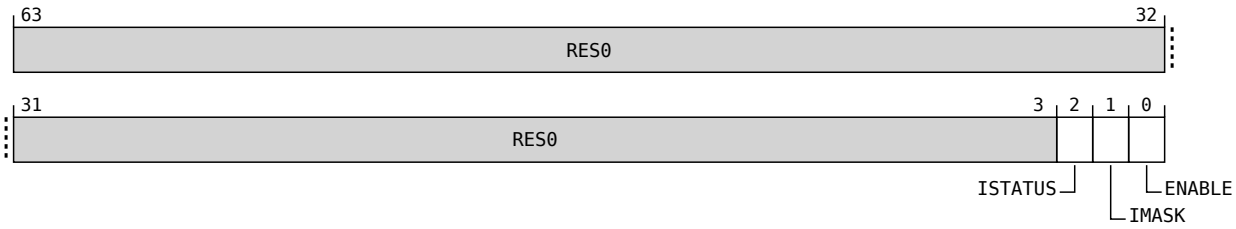
Configuration

There are no configuration notes.

Attributes

CNTV_CTL_EL0 is a 64-bit register.

Field descriptions



Bits [63:3]

Reserved, RES0.

ISTATUS, bit [2]

The status of the timer. This bit indicates whether the timer condition is met:

ISTATUS	Meaning
0b0	Timer condition is not met.
0b1	Timer condition is met.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0 then the timer interrupt is asserted.

When the value of the ENABLE bit is 0, the ISTATUS field is UNKNOWN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RO.

IMASK, bit [1]

Timer interrupt mask bit. Permitted values are:

IMASK	Meaning
0b0	Timer interrupt is not masked by the IMASK bit.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b1	Timer interrupt is masked by the IMASK bit.

For more information, see the description of the ISTATUS bit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

ENABLE, bit [0]

Enables the timer. Permitted values are:

ENABLE	Meaning
0b0	Timer disabled.
0b1	Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTV_TVAL_EL0 continues to count down.

Note

Disabling the output signal might be a power-saving option.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTV_CTL_EL0

Accesses to this register use the following encodings in the System register encoding space:

MRS <Xt>, CNTV_CTL_EL0

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0011	0b001

```
if PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1.EL0VTEN == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        X[t, 64] = CNTV_CTL_EL0;
elsif PSTATE.EL == EL1 then
    X[t, 64] = CNTV_CTL_EL0;
elsif PSTATE.EL == EL2 then
    X[t, 64] = CNTV_CTL_EL0;
```

MSR CNTV_CTL_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0011	0b001

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1.EL0VTEN == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        CNTV_CTL_EL0 = X[t, 64];
elsif PSTATE.EL == EL1 then
    CNTV_CTL_EL0 = X[t, 64];
elsif PSTATE.EL == EL2 then
    CNTV_CTL_EL0 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.8.12 CNTV_CVAL_EL0, Counter-timer Virtual Timer CompareValue Register

The CNTV_CVAL_EL0 characteristics are:

Purpose

Holds the compare value for the virtual timer.

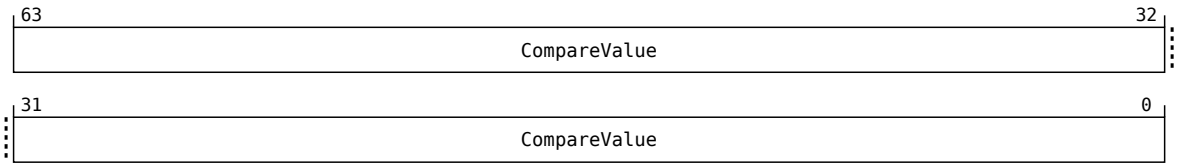
Configuration

There are no configuration notes.

Attributes

CNTV_CVAL_EL0 is a 64-bit register.

Field descriptions



CompareValue, bits [63:0]

Holds the EL1 virtual timer CompareValue.

When CNTV_CTL_EL0.ENABLE is 1, and TimerConditionMet is TRUE for the EL1 virtual timer, the timer condition is met and all of the following are true:

- CNTV_CTL_EL0.ISTATUS is set to 1.
- If CNTV_CTL_EL0.IMASK is 0, an interrupt is generated.

TimerConditionMet is defined by 'Operation of the CompareValue views of the timers'.

The CompareValue view of the timer acts like a 64-bit upcounter timer.

When CNTV_CTL_EL0.ENABLE is 0, the timer condition is not met, but CNTVCT_EL0 continues to count.

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTV_CVAL_EL0

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, CNTV_CVAL_EL0
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0011	0b010

```
if PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1.EL0VTEN == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        X[t, 64] = CNTV_CVAL_EL0;
elsif PSTATE.EL == EL1 then
    X[t, 64] = CNTV_CVAL_EL0;
elsif PSTATE.EL == EL2 then
    X[t, 64] = CNTV_CVAL_EL0;
```

MSR CNTV_CVAL_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0011	0b010

```
if PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1.EL0VTEN == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        CNTV_CVAL_EL0 = X[t, 64];
elsif PSTATE.EL == EL1 then
    CNTV_CVAL_EL0 = X[t, 64];
elsif PSTATE.EL == EL2 then
    CNTV_CVAL_EL0 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.8.13 CNTV_TVAL_EL0, Counter-timer Virtual Timer TimerValue Register

The CNTV_TVAL_EL0 characteristics are:

Purpose

Holds the timer value for the EL1 virtual timer.

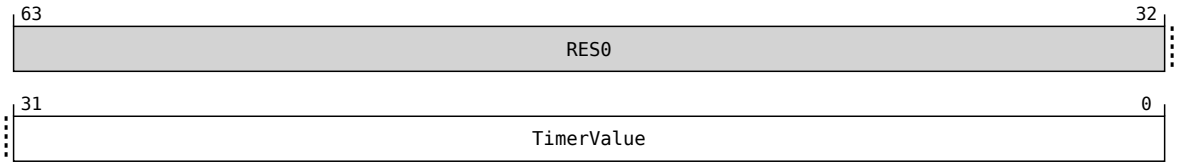
Configuration

There are no configuration notes.

Attributes

CNTV_TVAL_EL0 is a 64-bit register.

Field descriptions



Bits [63:32]

Reserved, RES0.

TimerValue, bits [31:0]

The TimerValue view of the EL1 virtual timer.

On a read of this register:

- If CNTV_CTL_EL0.ENABLE is 0, the value returned is UNKNOWN.
- If CNTV_CTL_EL0.ENABLE is 1, the value returned is (CNTV_CVAL_EL0 - CNTVCT_EL0).

On a write of this register, CNTV_CVAL_EL0 is set to (CNTVCT_EL0 + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When CNTV_CTL_EL0.ENABLE is 1, the timer condition is met when (CNTVCT_EL0 - CNTV_CVAL_EL0) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:

- CNTV_CTL_EL0.ISTATUS is set to 1.
- If CNTV_CTL_EL0.IMASK is 0, an interrupt is generated.

When CNTV_CTL_EL0.ENABLE is 0, the TimerValue cannot be read but continues to decrement. When the timer is enabled, the TimerValue represents the elapsed time whether that time was spent enabled or disabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTV_TVAL_EL0

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, CNTV_TVAL_EL0
```

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0011	0b000

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1.EL0VTEN == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif !ELIsInHost(EL0) then
        if CNTV_CTL_EL0.ENABLE == '0' then
            X[t, 64] = bits(64) UNKNOWN;
        else
            X[t, 64] = ZeroExtend((CNTV_CVAL_EL0 - (PhysicalCountInt() - CNTVOFF_EL2))<31:0>, 64);
    else
        if CNTV_CTL_EL0.ENABLE == '0' then
            X[t, 64] = bits(64) UNKNOWN;
        else
            X[t, 64] = ZeroExtend((CNTV_CVAL_EL0 - PhysicalCountInt())<31:0>, 64);
elsif PSTATE.EL == EL1 then
    if CNTV_CTL_EL0.ENABLE == '0' then
        X[t, 64] = bits(64) UNKNOWN;
    else
        X[t, 64] = ZeroExtend((CNTV_CVAL_EL0 - (PhysicalCountInt() - CNTVOFF_EL2))<31:0>, 64);
elsif PSTATE.EL == EL2 then
    if !ELIsInHost(EL2) then
        if CNTV_CTL_EL0.ENABLE == '0' then
            X[t, 64] = bits(64) UNKNOWN;
        else
            X[t, 64] = ZeroExtend((CNTV_CVAL_EL0 - (PhysicalCountInt() - CNTVOFF_EL2))<31:0>, 64);
    else
        if CNTV_CTL_EL0.ENABLE == '0' then
            X[t, 64] = bits(64) UNKNOWN;
        else
            X[t, 64] = ZeroExtend((CNTV_CVAL_EL0 - PhysicalCountInt())<31:0>, 64);
```

MSR CNTV_TVAL_EL0, <Xt>

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0011	0b000

```
if PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1.EL0VTEN == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif !ELIsInHost(EL0) then
        CNTV_CVAL_EL0 = (SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt()) - CNTVOFF_EL2;
    else
        CNTV_CVAL_EL0 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
elsif PSTATE.EL == EL1 then
    CNTV_CVAL_EL0 = (SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt()) - CNTVOFF_EL2;
elsif PSTATE.EL == EL2 then
    if !ELIsInHost(EL2) then
        CNTV_CVAL_EL0 = (SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt()) - CNTVOFF_EL2;
    else
        CNTV_CVAL_EL0 = SignExtend(X[t, 64]<31:0>, 64) + PhysicalCountInt();
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.8.14 CNTVCT_EL0, Counter-timer Virtual Count Register

The CNTVCT_EL0 characteristics are:

Purpose

Reads of CNTVCT_EL0 return the 64-bit physical count value minus a virtual offset.

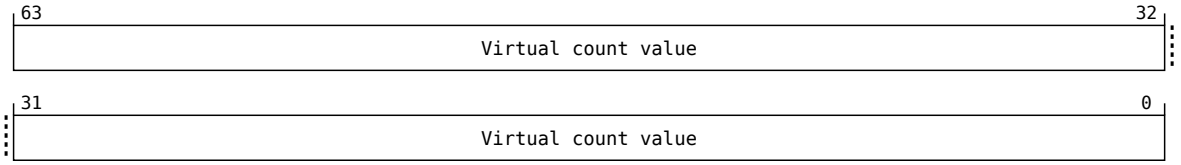
Configuration

There are no configuration notes.

Attributes

CNTVCT_EL0 is a 64-bit register.

Field descriptions



VirtualCount, bits [63:0]

Virtual count value.

Reads of CNTVCT_EL0 return PhysicalCountInt<63:0>.

PhysicalCountInt is defined by 'The Physical Counter'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTVCT_EL0

Accesses to this register use the following encodings in the System register encoding space:

`MRS <Xt>, CNTVCT_EL0`

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0000	0b010

```
if PSTATE.EL == EL0 then
    if !ELIsInHost(EL0) && CNTKCTL_EL1.EL0VCTEN == '0' then
        if HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    else
        if !ELIsInHost(EL0) then
            X[t, 64] = PhysicalCountInt() - CNTVOFF_EL2;
        else
            X[t, 64] = PhysicalCountInt();
elsif PSTATE.EL == EL1 then
    X[t, 64] = PhysicalCountInt() - CNTVOFF_EL2;
elsif PSTATE.EL == EL2 then
    if !ELIsInHost(EL2) then
        X[t, 64] = PhysicalCountInt() - CNTVOFF_EL2;
    else
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        X[t, 64] = PhysicalCountInt();
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

D12.8.15 CNTVOFF_EL2, Counter-timer Virtual Offset Register

The CNTVOFF_EL2 characteristics are:

Purpose

Holds the 64-bit virtual offset. This is the offset for the AArch64 virtual timers and counters.

Configuration

The virtual offset applies to:

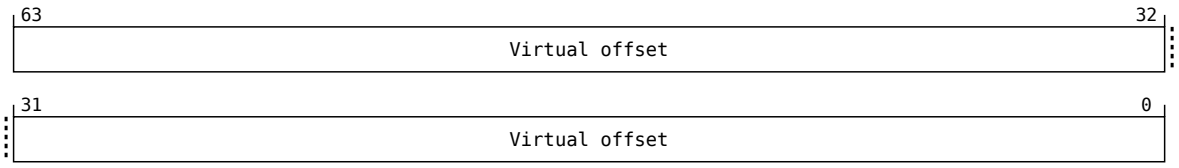
- Direct reads of CNTVCT_EL0.
- Indirect reads of the virtual counter by the EL1 virtual timer. See 'The Generic Timer in AArch64 state'.
- Indirect reads of the virtual counter for timestamps generated by profiling logic. See 'AArch64 Self-hosted Trace'.

The virtual offset only applies under conditions described by the relevant sections.

Attributes

CNTVOFF_EL2 is a 64-bit register.

Field descriptions



VOffset, bits [63:0]

Virtual offset.

If the Generic counter is implemented at a size less than 64 bits, then this field is permitted to be implemented at the same width as the counter, and the upper bits are RES0.

The value of this field is treated as zero-extended in all counter calculations.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTVOFF_EL2

Accesses to this register use the following encodings in the System register encoding space:

```
MRS <Xt>, CNTVOFF_EL2
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0000	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    X[t, 64] = CNTVOFF_EL2;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MSR CNTVOFF_EL2, <Xt>
```

op0	op1	CRn	CRm	op2
0b11	0b100	0b1110	0b0000	0b011

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    CNTVOFF_EL2 = X[t, 64];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Part E
External Debug

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter E1
About External Debug

This chapter gives an overview of A-profile external debug and specifies the required debug authentication. It contains the following sections:

- Introduction to external debug.
- External debug.
- Required debug authentication.

Note

For information about self-hosted debug, see AArch64 Self-hosted Debug.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E1.1 Introduction to external debug

The R-profile supports both:

Self-hosted debug

The PE itself hosts a debugger. That is, developers developing software to run on the PE use debugger software running on the same PE.

External debug

The debugger is external to the PE. The debugging might be either on-chip, for example in a second PE, or off-chip, for example a JTAG debugger that accesses the chip through a Debug Access Port.

External debug is particularly useful for:

- Hardware bring-up. That is, debugging during development when a system is first powered up and not all of the software functionality is available.
- PEs that are deeply embedded inside systems.

To support external debug, the Arm architecture defines required features that are called *external debug features*.

Note

An external debugger has a potentially high level of control over and visibility into the PE. The system sets this level using debug authentication. See Required debug authentication.
If the debug authentication level is set too low, agents may be able to bypass elements of the security and privilege models. This includes both off-chip agents and on-chip agents such as unprivileged or Non-secure software.

E1.1.1 Definition and constraints of a debugger in the context of external debug

When the description of external debug in this Part of the manual describes a *debugger* as controlling external debug this debugger might be a second on-chip PE or an off-chip device such as a JTAG debugger using a Debug Access Port (DAP).

If a Debug Access Port is implemented:

- When debug is prohibited at the Debug Access Port, the port must not generate accesses to the external debug interface of the PE.
- When debug is prohibited for a Security state at the Debug Access Port, it must not be possible to generate accesses to PA spaces that are not accessible for that Security state.
- When accesses for a Security state are allowed at the Debug Access Port, the port must be able to generate accesses to PA spaces that are accessible for that Security state.

If FEAT_Debugv8p4 is not implemented, accesses to the PE are controlled by the external authentication interface functions, `ExternalInvasiveDebugEnabled()`, `ExternalNoninvasiveDebugEnabled()`, `ExternalSecureNoninvasiveDebugEnabled()`, and `ExternalSecureInvasiveDebugEnabled()`.

The Debug Access Port is not required to use the same authentication interface as the PE.

Arm recommends that when `ExternalSecureInvasiveDebugEnabled()` == FALSE at the PE, access through the DAP is disabled.

See also Required debug authentication.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E1.2 External debug

Debug events allow an external debugger to halt the PE. The R-profile provides the following debug events:

- Halting Step debug events:
  - The debugger can use this resource to make the PE step through code one line at a time.
- Halt Instruction debug event:
  - This might occur when software executes the Halting breakpoint instruction, `HLT`.
- Exception Catch debug event:
  - This can be programmed to occur on all entries to a given Exception level.
- External Debug Request debug event:
  - An embedded cross-trigger can signal this debug event.
- OS Unlock Catch debug event:
  - This might occur when the state of the OS Lock changes from locked to unlocked.
- Reset Catch debug events:
  - This might occur when the PE exits reset state.
- Software Access debug event:
  - This can be programmed to occur when software tries to access the Breakpoint Value registers, the Breakpoint Control registers, the Watchpoint value registers, or the Watchpoint Control registers. It caused a trap to Debug state.

Breakpoints and watchpoints can also halt the PE.

When the PE is in Debug state:

- It stops executing instructions from the location indicated by the program counter, and is instead controlled through the external debug interface.
- The *Instruction Transfer Register,* ITR, passes instructions to the PE to execute in Debug state:
  - The ITR contains a single register, EDITR, and associated flow-control flags.
- The *Debug Communications Channel*, DCC, passes data between the PE and the debugger:
  - The DCC includes the data transfer registers, DTRRX and DTRTX, and associated flow-control flags.
  - Although the DCC is an essential part of Debug state operation, it can also be used in Non-debug state.
- The PE cannot service any interrupts in Debug state.

Debug State describes Debug state in more detail.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E1.3 Required debug authentication

Any implementation must provide the debug authentication defined in this section, which controls:

- Whether the PE can halt.
- Whether some aspects of non-invasive debug are permitted.

The pseudocode functions shown in , and the conditions that follow that table, define the architectural requirements for debug authentication.

Table E1-1 Debug authentication functions

Pseudocode function	Description
`ExternalNoninvasiveDebugEnabled()`	Returns TRUE if Non-secure non-invasive debug is enabled.
`ExternalInvasiveDebugEnabled()`	Returns TRUE if Non-secure invasive debug is enabled.
`ExternalSecureNoninvasiveDebugEnabled()`	Returns TRUE if Secure non-invasive debug is enabled.
`ExternalSecureInvasiveDebugEnabled()`	Returns TRUE if Secure invasive debug is enabled.

The following conditions always apply:

- If `ExternalInvasiveDebugEnabled()` is FALSE then `ExternalSecureInvasiveDebugEnabled()` is FALSE.
- If `ExternalNoninvasiveDebugEnabled()` is FALSE then `ExternalSecureNoninvasiveDebugEnabled()` is FALSE.
- If `ExternalInvasiveDebugEnabled()` is TRUE then `ExternalNoninvasiveDebugEnabled()` is TRUE.
- If `ExternalSecureInvasiveDebugEnabled()` is TRUE then `ExternalSecureNoninvasiveDebugEnabled()` is TRUE.

If FEAT_Debugv8p4 is implemented:

- `ExternalNoninvasiveDebugEnabled()` always returns TRUE.
- `ExternalSecureNoninvasiveDebugEnabled()` returns the same as `ExternalSecureInvasiveDebugEnabled()`.

Arm recommends the use of the interface described in Recommended authentication interface to provide this debug authentication. The pseudocode functions in A-profile Architecture Pseudocode, which are linked to by the entries in the Pseudocode function column of , assume that this interface is implemented.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter E2
Debug State

This chapter describes Debug state. It contains the following sections:

- About Debug state.
- Halting the PE on debug events.
- Entering Debug state.
- Behavior in Debug state.
- Exiting Debug state.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E2.1 About Debug state

In external debug, debug events allow an external debugger to halt the PE. The PE then enters Debug state. When the PE is in Debug state:

- It stops executing instructions from the location indicated by the Program Counter, and is instead controlled through the external debug interface.
- The *Instruction Transfer Register,* ITR, passes instructions to the PE to execute in Debug state.
- The *Debug Communications Channel*, DCC, passes data between the PE and the debugger.

The PE cannot service any interrupts in Debug state.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E2.2 Halting the PE on debug events

For details of debug events, see Introduction to Halting debug events and Breakpoint and Watchpoint debug events.

On a debug event, the PE must do one of the following:

- Enter Debug state.
- Pend the debug event.
- Generate a debug exception.
- Ignore the debug event.

This behavior depends on both:

- Whether halting is allowed by the current state of the debug authentication interface. See Halting allowed and halting prohibited.
- The type of debug event and the programming of the debug control registers.
  - See Halting debug events for all Halting debug events.
  - See Breakpoint and Watchpoint debug events for Breakpoint and Watchpoint debug events.

See also Other debug exceptions.

This means that behavior can be CONSTRAINED UNPREDICTABLE if the conditions change. See Synchronization and Halting debug events.

Table E3-1 summarizes the possible outcomes of each type of debug event.

E2.2.1 Halting allowed and halting prohibited

Halting can be either allowed or prohibited:

- Halting is always prohibited in Debug state.
- Halting is always prohibited when `DoubleLockStatus()` == TRUE.
  - This means that FEAT_DoubleLock is implemented and OS Double lock is locked.
- Halting is also controlled by the IMPLEMENTATION DEFINED authentication interface, and is prohibited when `ExternalSecureInvasiveDebugEnabled()` == FALSE.
- Otherwise, halting is allowed.

For more information, see:

- Pseudocode description of Halting on debug events.
- Required debug authentication.
- Recommended External Debug Interface.

E2.2.2 Halting debug events

The Halting debug events are described in Halting Debug Events.

When a Halting debug event is generated, it causes entry to Debug state if all of the following are true:

- Halting is allowed. See Halting allowed and halting prohibited.
- The Halting debug event is one of:
  - A Halt Instruction debug event and EDSCR.HDE == 1.
  - A Software Access debug event and OSLSR_EL1.OSLK == 0, meaning that the OS Lock is unlocked.
  - Neither a Halt Instruction debug event nor a Software Access debug event.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

— A Halt Instruction debug event is the only Halting debug event that relies on EDSCR.HDE == 1.

— Halting on Breakpoint and Watchpoint debug events is also controlled by EDSCR.HDE. See Breakpoint and Watchpoint debug events.

— EDSCR.HDE can be written by software when the OS Lock is locked. Privileged code can use MDCR_EL2, Monitor Debug Configuration Register (EL2).TDOSA to trap writes to OS Lock registers.

If a Halting debug event does not generate entry to Debug state because the conditions listed in this section do not hold, then:

- If the Halting debug event is a Halt Instruction debug event, the instruction that generated the Halting debug event is treated as UNDEFINED.
- If the Halting debug event is a Software Access debug event, it is ignored.
- If the Halting debug event is an Exception Catch debug event, it is ignored except when FEAT_Debugv8p8 is implemented, in which case the event might be pended.

In all other cases the Halting debug event is pended, see Pending Halting debug events.

Summary of actions from debug events summarizes the possible outcome for each type of Debug event.

Note

Halting debug events never generate debug exceptions.

E2.2.3 Breakpoint and Watchpoint debug events

A breakpoint or watchpoint generates an entry to Debug state if all of the following conditions hold:

- Halting debug is enabled, that is EDSCR.HDE == 1.
- Halting is allowed. See Halting allowed and halting prohibited.
- The OS Lock is unlocked, that is OSLSR_EL1, OS Lock Status Register.OSLK == 0.

The Address Mismatch breakpoint type is reserved when all of these conditions are met.

MDSCR_EL1.MDE is ignored when determining whether to enter Debug state. A breakpoint or watchpoint that generates entry to Debug state is a Breakpoint or Watchpoint debug event and does not generate a debug exception.

A breakpoint or watchpoint that does not generate an entry to Debug state either:

- Generates a Breakpoint or Watchpoint exception.
- Is ignored.

Note

EDSCR.HDE is ignored when determining whether to generate a debug exception. The debug exception is suppressed only if the PE enters Debug state. This means that the use of Halting debug mode in one Security state does not affect the Exception model in another Security state.

See AArch64 Self-hosted Debug, and the Note in Other debug exceptions.

E2.2.4 Other debug exceptions

The following events never generate entry to Debug state:

- Breakpoint Instruction exceptions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Software Step exceptions.

The behavior of these events is unchanged when Halting debug mode is enabled, that is when EDSCR.HDE == 1. This means that these events can do one of the following:

- They can generate a debug exception.
- They can be ignored.

For additional information, see AArch64 Self-hosted Debug.

E2.2.5 Debug state entry and debug event prioritization

The following are synchronous debug events:

- Breakpoint debug event.
- Watchpoint debug event.
- Halting Step debug event.
- Halt Instruction debug event.
- Exception Catch debug event.
- Software Access debug event.
- Reset Catch debug event.

Each of these synchronous debug events are treated as a synchronous exception generated by an instruction, or by the taking of an exception or reset. That is, if halting is allowed, the synchronous debug event must be taken before any subsequent instructions are executed. Reset Catch debug events must be taken before the PE executes the instruction at the reset vector.

Note

- Reset Catch and Exception Catch debug events might be generated asynchronously, because they can result from an asynchronous exception. However, if halting is allowed after the reset or asynchronous exception has been processed, the Reset Catch or Exception Catch debug event is taken synchronously.
- The Halting Step debug event is generated by the instruction after the stepped instruction. Therefore, if the stepped instruction generates any other synchronous exceptions or debug events these are taken first.

If halting is prohibited then Halting Step debug events and Reset Catch debug events might be pended and taken asynchronously. OS Unlock Catch debug events are always pended and taken asynchronously. See Pending Halting debug events.

The architecture does not define when asynchronous debug events are taken, and therefore the prioritization of asynchronous debug events is IMPLEMENTATION DEFINED. See Synchronization and Halting debug events.

The following list shows how the synchronous debug events are prioritized, with 1 being the highest priority.

Note

The priority numbering is the same as the numbering for AArch64 synchronous exception priorities listed in Synchronous exception types, and in particular Prioritization of interrupts.

The priority for synchronous debug events is as follows:

1 Reset Catch debug event. See Reset Catch debug events.

This debug event has a higher priority than the synchronous exceptions listed in Synchronous exception types.

2 Exception Catch debug event if it has a priority of 2. See Exception Catch debug event.

This debug event can be assigned one of two priorities, 2 or 6. When it has a priority of 2, it has a higher priority than the synchronous exceptions listed in the Exception model.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3	Halting Step debug event. See Halting Step debug events. This debug event has a higher priority than the synchronous exceptions listed in the Exception model.
4-5	These events are not debug events.
6	Exception Catch debug event if it has a priority of 6. See Exception Catch debug event. This debug event can be assigned one of two priorities, 2 or 6.
7-9	These events are not debug events.
10	Breakpoint debug events. See Breakpoint and Watchpoint debug events.
11-12	These events are not debug events.
13	Halt Instruction debug event. See Halt Instruction debug event.
14-39	These events are not debug events.
40	Software Access debug event. See Software Access debug event.
41-45	These events are not debug events.
46	Watchpoint debug events. See Breakpoint and Watchpoint debug events.
47-48	These events are not debug events.

For Reset Catch debug events and Halting Step debug events, the priorities listed in this section apply only when halting is allowed at the time the event is generated. This means that the event is taken synchronously and not pended.

For more information on the prioritization of exceptions, see:

- Synchronous exception types.
- Prioritization of interrupts.

E2.2.6 Imprecise entry to Debug state

Debug state entry is normally *precise*. This means that the PE cannot enter Debug state if it can neither complete nor abandon all currently executing instructions and leave the PE in a precise state. See Definition of a precise exception and imprecise exception.

A debugger can write a value of 1 to EDRCR.CBRRQ to allow imprecise entry to Debug state. An External Debug Request debug event must be pending before writing 1 to this bit. Support for this feature is OPTIONAL and it is IMPLEMENTATION DEFINED when it is effective at forcing entry to Debug state.

The PE ignores writes to this bit when `ExternalInvasiveDebugEnabled()` == FALSE.

Example E2-1 shows how entry to Debug state can be forced.

Example E2-1 Forcing entry to Debug state

The debugger pends an External Debug Request debug event through the CTI to halt a program that has stopped responding. However, the memory system is not responding and a memory access instruction cannot complete. This means that Debug state cannot be entered precisely. The debugger writes a value of 1 to EDRCR.CBRRQ. The PE cancels all outstanding memory accesses and enters Debug state. As some instructions might not have completed correctly, entry to Debug state is imprecise.

When Debug state is entered imprecisely, all memory access instructions executed through the ITR have CONSTRAINED UNPREDICTABLE behavior. The value of all registers is UNKNOWN, but might be useful for diagnostic purposes.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E2.2.7 Summary of actions from debug events

Table E2-1 shows the Software and Halting debug events. In Table E2-1, the columns have the following meaning:

Debug event type

This means the type of debug event where:

Other software Means one of:

- Software Step exceptions.
- Breakpoint Instruction exceptions.

Other Halting Means one of the following:

- Halting Step debug events.
- External Debug Request debug event.
- Reset Catch debug events.
- OS Unlock Catch debug event.

Other debug events are referred to explicitly.

Authentication

This means halting is allowed by the IMPLEMENTATION DEFINED external authentication interface. It is the result of `ExternalSecureInvasiveDebugEnabled()`.

DLK This indicates whether FEAT_DoubleLock is implemented and locked, `DoubleLockStatus()` == TRUE.

OSLK This is the value of OSLSR_EL1, OS Lock Status Register.OSLK. It indicates whether the OS Lock is locked.

HDE This is the value of EDSCR.HDE. It indicates whether Halting debug is enabled.

The letter X in Table E2-1 indicates that the value can be either 0 or 1.

Table E2-1 Debug authentication for external debug

Debug event type	Authentication	DLK	OSLK	HDE	Behavior
Other software	X	X	X	X	Handled by the Exception model
Breakpoint or Watchpoint debug event	X	TRUE	X	X	Handled by the Exception model (ignored)
	X	FALSE	1	X	Handled by the Exception model (ignored)
	FALSE	FALSE	0	X	Handled by the Exception model
	TRUE	FALSE	0	0	Handled by the Exception model
	TRUE	FALSE	0	1	Entry to Debug state
Halt Instruction debug event	FALSE	X	X	X	UNDEFINED
	TRUE	TRUE	X	X	UNDEFINED
	TRUE	FALSE	X	0	UNDEFINED
	TRUE	FALSE	X	1	Entry to Debug state
Exception Catch debug event	FALSE	X	X	X	Might be pended[a]

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Debug event type	Authentication	DLK	OSLK	HDE	Behavior
	TRUE	TRUE	X	X	Ignored
	TRUE	FALSE	X	X	Entry to Debug state
Software Access debug event	FALSE	X	X	X	Ignored
	TRUE	TRUE	X	X	Ignored
	TRUE	FALSE	1	X	Ignored
	TRUE	FALSE	0	X	Entry to Debug state
Other Halting	FALSE	X	X	X	Debug event is pended
	TRUE	TRUE	X	X	Debug event is pended
	TRUE	FALSE	X	X	Entry to Debug state

a. See Exception Catch debug event.

E2.2.8 Pseudocode description of Halting on debug events

The `Halted()`, `Restarting()`, `HaltingAllowed()`, and `HaltOnBreakpointOrWatchpoint()` functions are described in the pseudocode.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E2.3 Entering Debug state

On entry to Debug state, the preferred restart address and PSTATE are saved in DLR_EL0, Debug Link Register and DSPSR_EL0, Debug Saved Program Status Register. The PE remains in the mode and Security state from which it entered Debug state.

If EDRCR.CBRRQ has a value of 0, entry to Debug state is precise. If EDRCR.CBRRQ has a value of 1, then imprecise entry to Debug state is permitted.

If a Watchpoint debug event causes an entry to Debug state, the PE records all of the following:

- When FEAT_EDHSR is not implemented or the PE sets EDHSR.FnV to 0, the address of the access that generated the Watchpoint debug event is recorded in EDWAR.
- When FEAT_EDHSR is implemented, information about the access that generated the Watchpoint debug event is recorded in EDHSR.

For more information, see:

- Exception syndrome information, fault address information, and preferred return address for a debug event taken from AArch64 state.

Other than the effect on PSTATE and EDSCR, entry to Debug state is not a Context Synchronization event. The effects of entry to Debug state on PSTATE and EDSCR are synchronized.

E2.3.1 Effect of Debug state entry on DLR and DSPSR

DLR_EL0, Debug Link Register is set to the preferred restart address for the debug event, which depends on the event type. The value of PSTATE is saved in DSPSR_EL0, Debug Saved Program Status Register.

For synchronous Halting debug events, the preferred restart address is the address of the instruction that generated the debug event.

For asynchronous Halting debug events, including pending Halting debug events taken asynchronously, the preferred restart address is the address of the first instruction that must be executed on exit from Debug state.

This means that:

- For Breakpoint and Watchpoint debug events, the preferred restart address is the same as the preferred return address for a debug exception, as described in AArch64 Self-hosted Debug.
- For Halt Instruction debug events, DLR_EL0, Debug Link Register is set to the address of the `HLT` instruction and DSPSR_EL0, Debug Saved Program Status Register.IT is correct for the `HLT` instruction.
- For Software Access debug events, DLR_EL0, Debug Link Register is set to the address of the accessing instruction and DSPSR_EL0, Debug Saved Program Status Register.IT is correct for this instruction.
- For Halting Step debug events taken synchronously, DLR_EL0, Debug Link Register and DSPSR_EL0, Debug Saved Program Status Register are set as the ELR and SPSR would be set for a Software Step exception. This is usually the address of, and PSTATE for, the instruction after the one that was stepped.
- For Exception Catch debug events, DLR_EL0, Debug Link Register is set to the address of the first instruction that must be executed on exit from Debug state, and DSPSR_EL0, Debug Saved Program Status Register is correct for this instruction. This is the exception vector or reset vector address.
  - If the debug event is generated on taking an exception to a trapped Exception level, the DLR_EL0, Debug Link Register is set to the address of the exception vector the PE would have started fetching from. This is UNKNOWN if the VBAR for the Exception level has never been initialized. The DSPSR_EL0, Debug Saved Program Status Register records the value of PSTATE after taking the exception. The Exception Catch occurs after the SPSR and the Link register are set, and the debugger can use these registers to determine where in the application program the exception occurred.

    Note

    Depending on the target Exception level and Execution state for the exception, the Link register is one of ELR_EL1, ELR_EL2 or LR (R14).

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— If the debug event is generated on an exception return to a trapped Exception level, the DLR_EL0, Debug Link Register is set to the target address of the exception return and the DSPSR_EL0, Debug Saved Program Status Register records the value of PSTATE after the exception return.

- Reset Catch debug events taken synchronously behave like Exception Catch debug events.
- For Reset Catch debug events and Exception Catch debug events generated on reset to a trapped Exception level, the DLR_EL0, Debug Link Register is set to is set to the reset address and the DSPSR_EL0, Debug Saved Program Status Register records the reset value of PSTATE.
- For pending Halting debug events and External Debug Request debug events, DLR_EL0, Debug Link Register is set to the address of the first instruction that must be executed on exit from Debug state. See Pending Halting debug events.

Normally DLR_EL0, Debug Link Register is aligned according to the instruction set state indicated in DSPSR_EL0, Debug Saved Program Status Register. However, a debug event might be taken at a point where the PC is not aligned.

E2.3.2 Effect of Debug state entry on System registers, the Event register, and Exclusives monitors

Entering Debug state has no effect on System registers other than DLR_EL0, Debug Link Register and DSPSR_EL0, Debug Saved Program Status Register. In particular, ESRs, FARs, and FSRs are not updated on entering Debug state.

Entering Debug state has no architecturally-defined effect on the Event Register and Exclusives monitors.

Note

Entry to Debug state might set the Event Register or clear the Exclusives monitors, or both. However, this is not a requirement, and debuggers must not rely on any IMPLEMENTATION SPECIFIC behavior.

Unless otherwise described in this reference manual, instructions executed in Debug state have their architecturally-defined effects on the System registers, the Event register, and Exclusives monitors.

E2.3.3 Effect of entering Debug state on PSTATE

The effect of an entry to Debug state on PSTATE is described in Entering Debug state.

On entry to Debug state after PSTATE is saved in DSPSR_EL0, Debug Saved Program Status Register:

- PSTATE.IL is cleared to 0.
- PSTATE.{SS, D, A, I, F, SSBS} are set to UNKNOWN values.

PSTATE.{N, Z, C, V, Q, M, nRW, EL, SP, PAN, DIT} are unchanged.

For more information, see PSTATE in Debug state.

E2.3.4 Entering Debug state during loads and stores

The PE can enter Debug state during instructions that perform a sequence of memory accesses, as opposed to a single single-copy atomic access, because of a Watchpoint debug event. The effect of entering Debug state on such an instruction is the same as taking a Data Abort exception during such an instruction.

In addition, the PE can enter Debug state because of an External Debug Request debug event during instructions that perform a sequence of memory accesses. The effect of entering Debug state on such an instruction is the same as taking an interrupt exception during such an instruction.

This applies to all memory types.

E2.3.5 Entering Debug state and Software Step

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When Software Step is active, a debug event that causes entry to Debug state behaves like an exception taken to an Exception level above the debug target Exception level. That is:

- If the instruction that is stepped generates a synchronous debug event that causes entry to Debug state, or an asynchronous debug event is taken before the step completes, the PE enters Debug state with DSPSR_EL0, Debug Saved Program Status Register.SS set to 1.
- A pending Halting debug event or an asynchronous debug event can be taken after the step has completed. In this case the PE enters Debug state with DSPSR_EL0, Debug Saved Program Status Register.SS set to 0.

In addition:

- If the instruction that is stepped generates an exception trapped by an Exception Catch debug event, the PE enters Debug state at the exception vector with DSPSR_EL0, Debug Saved Program Status Register.SS set to 0. This is because PSTATE.SS is set to 0 by taking the exception.
- If the PE is reset, PSTATE.SS is reset to 0. If the following debug events are enabled, and halting is allowed, the PE enters Debug state with DSPSR_EL0, Debug Saved Program Status Register.SS set to 0:
  - Reset Catch debug event at the reset Exception level.
  - Exception Catch debug event at the reset Exception level.
  - Halting Step debug event.
- If Halting Step is also active, then Halting Step and Software Step operate in parallel and can both become active-pending. In this case Halting step has a higher priority than Software step. This means that the PE enters Debug state and DSPSR_EL0, Debug Saved Program Status Register.SS is set to 0.

E2.3.6 Pseudocode description of entering Debug state

The `DebugHalt` constants are described in shared/debug/halting/DebugHalt in the pseudocode. The `UpdateEDSCRFields()` and `Halt()` functions are described in A-profile Architecture Pseudocode.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E2.4 Behavior in Debug state

Instructions are executed in Debug state from the Instruction Transfer Register, ITR. The debugger controls which instructions are executed in Debug state by writing the instructions to the External Debug Instruction Transfer register, EDITR.

Some instructions are available only in Debug state. See Debug state operations, DCPS, DRPS, MRS, MSR. In Non-debug state these instructions are UNDEFINED.

The following sections describe behavior in Debug state:

- PSTATE in Debug state.
- Executing instructions in Debug state.
- Decode tables.
- Security in Debug state.
- Privilege in Debug state.
- Debug state operations, DCPS, DRPS, MRS, MSR.
- Exceptions in Debug state.
- Accessing registers in Debug state.
- Accessing memory in Debug state.

This section specifies the CONSTRAINED UNPREDICTABLE behaviors that apply in Debug state, but see Changing the value of EDECR.SS when not in Debug state for a change in Non-debug state that causes CONSTRAINED UNPREDICTABLE behavior.

E2.4.1 PSTATE in Debug state

PSTATE.{N, Z, C, V, SS, ALLINT, SSBS, BTYPE, D, A, I, F, PM}{N, Z, C, V, Q, SS, SSBS, D, A, I, F, PM} are all ignored in Debug state:

- There are no conditional instructions in Debug state.
- Software step is inactive.
- Asynchronous exceptions and debug events are ignored.
- If FEAT_SSBS is implemented, then hardware is permitted to load or store speculatively, regardless of the value of PSTATE.SSBS.

Instructions executed in Debug state indirectly read PSTATE.{PAN, IL, nRW, EL, SP} as they would in Non-debug state.

Note

PSTATE.DIT is not guaranteed to have any effect in Debug state.

In Debug state:

- PSTATE.PAN is set to 1 by:
  - A `DCPS<n>` instruction to EL1 using AArch64 if SCTLR_EL1.SPAN == 0.
  - A `DCPS<n>` instruction to EL2 using AArch64 if SCTLR_EL2.SPAN == 0.
- PSTATE.UAO is set to 0 by a `DCPS<n>` instruction to AArch64 state.
- If FEAT_SSBS is implemented, the `DCPS<n>` instruction leaves the PSTATE.SSBS bit UNKNOWN.

PSTATE can also be changed by taking exceptions in Debug state, and by the execution of `DCPS<n>` and `DRPS` instructions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E2.4.2 Executing instructions in Debug state

The instructions executed in Debug state must be either A64 instructions, depending on the current Execution state.

Each instruction falls into one of the following groups:

- Debug state instructions. These are instructions that are changed in Debug state. See A64 instructions that are changed in Debug state.
- Instructions that are unchanged in Debug state. See A64 instructions that are unchanged in Debug state. Where these sections describe the behavior of an instruction as unchanged in Debug state, or having the same behavior as in Non-debug state, and that instruction would generate an exception in Non-debug state, then the behavior of the exception is as described in Exceptions in Debug state.
- Instructions that are UNPREDICTABLE or CONSTRAINED UNPREDICTABLE in Debug state. See A64 instructions that are CONSTRAINED UNPREDICTABLE in Debug state.

If EDSCR.SDD == 1, then an exception from Non-secure state is never taken to Secure state.

See Security in Debug state.

E2.4.2.1 Executing A64 instructions in Debug state

The following sections describe the behavior of the A64 instructions in Debug state:

- A64 instructions that are changed in Debug state.
- A64 instructions that are unchanged in Debug state.
- A64 instructions that are CONSTRAINED UNPREDICTABLE in Debug state.

E2.4.2.1.1 A64 instructions that are changed in Debug state

The following instructions are defined in Debug state, but are UNDEFINED in Non-debug state:

- `DCPS`.

  Note

  `DCPS` can be UNDEFINED in certain conditions in Debug state. See DCPS<n>.

- `DRPS`.
- `MRS` (DLR_EL0), `MRS` (DSPSR_EL0), `MSR` (DLR_EL0), `MSR` (DSPSR_EL0)

For more information, see Debug state operations, DCPS, DRPS, MRS, MSR.

E2.4.2.1.2 A64 instructions that are unchanged in Debug state

The following list shows the instructions that are unchanged in Debug state:

Any instruction that is UNDEFINED in Non-debug state

This list of instructions excludes:

- Any instruction listed in A64 instructions that are changed in Debug state.
- Any instruction listed in A64 instructions that are CONSTRAINED UNPREDICTABLE in Debug state that is UNDEFINED because an enable or disable bit is not RES0 or RES1

Instructions that move System or Special-purpose registers to or from a general-purpose register

This list of instructions:

- Includes the instructions to transfer a general-purpose register to or from the DTR, which can be executed at any Exception level.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Excludes PSTATE access instructions.

These instructions are:

- `MRS <special_reg>`, `MSR <special_reg>`.

  Note

  This does not include NZCV, DAIF, DAIFSet, DAIFClr, SPSel, CurrentEL, PAN, UAO and DIT.

- `MRS <system_reg>`, `MSR <system_reg>`.

Floating-point moves between a SIMD&FP register and a general-purpose register

These instructions are:

- `FMOV` (between a general-purpose register and a half-precision register).
- `FMOV` (between a general-purpose register and a single-precision register).
- `FMOV` (between a general-purpose register and a double-precision register).
- `FMOV` (between a general-purpose register and a SIMD element).

SIMD moves between a SIMD&FP register and a general-purpose register

These instructions are:

- `INS` (from a general-purpose register to a SIMD element).
- `UMOV` (from a SIMD element to a general-purpose register).

Barriers These instructions are:

- `DMB`.
- `DSB`.
- `ISB`.
- `CSDB`.
- `SSBB`.
- `PSSBB`.
- If FEAT_SB is implemented, then `SB`.

Memory access instructions at various access sizes

The following constraints apply:

- General purpose-registers only.
- One of the following addressing modes:
  - Unscaled (9-bit signed) immediate offset.
  - Immediate (9-bit signed) post-indexed.
  - Immediate (9-bit signed) pre-indexed.
  - Unprivileged (9-bit signed).
- Not literal.
- One of the following types:
  - (Single) register.
  - Exclusive.
  - Exclusive pair.
  - Acquire/Release.
  - Acquire/Release Exclusive.
  - Acquire/Release Exclusive pair.
- 32-bit and 64-bit target register variants.

These instructions are:

- `LDR`, `LDRB`, `LDRH`, `LDRSB`, `LDRSH`, `LDRSW` (immediate, not literal).

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- `LDUR`, `LDURB`, `LDURH`, `LDURSB`, `LDURSH`, `LDURSW` (immediate).
- `LDTR`, `LDTRB`, `LDTRH`, `LDTRSB`, `LDTRSH`, `LDTRSW` (immediate).
- `LDAR`, `LDARB`, `LDARH`, `LDXR`, `LDXRB`, `LDXRH`, `LDAXR`, `LDAXRB`, `LDAXRH`.
- `LDXP`, `LDAXP`.
- `STR`, `STRB`, `STRH` (immediate).
- `STUR`, `STURB`, `STURH` (immediate).
- `STTR`,`STTRB`, `STTRH` (immediate).
- `STLR`, `STLRB`, `STLRH`, `STXR`, `STXRB`, `STXRH`, `STLXR`, `STLXRB`, `STLXRH`.
- `STXP`, `STLXP`.
- If FEAT_LOR is implemented, then:
  - `LDLAR`, `LDLARB`, `LDLARH`.
  - `STLLR`, `STLLRBB`, `STLLRH`.
- If FEAT_LSE is implemented, then:
  - `CAS`, `CASB`, `CASH`, `CASP`.
  - `SWP`, `SWPB`, `SWPH`.
  - `LDADD`, `LDADDB`, `LDADDH`.
  - `LDCLR`, `LDCLRB`, `LDCLRH`.
  - `LDEOR`, `LDEORB`, `LDEORH`.
  - `LDSET`, `LDSETB`, `LDSETH`.
  - `LDSMAX`, `LDSMAXB`, `LDSMAXH`.
  - `LDSMIN`, `LDSMINB`, `LDSMINH`.
  - `LDUMAX`, `LDUMAXB`, `LDUMAXH`.
  - `LDUMIN`, `LDUMINB`, `LDUMINH`.
  - `STADD`, `STADDB`, `STADDH`.
  - `STCLR`, `STCLB`, `STCLRH`.
  - `STEOR`, `STEORB`, `STEORH`.
  - `STSET`, `STSETB`, `STSETH`.
  - `STSMAX`, `STSMAXB`, `STSMAXH`.
  - `STSMIN`, `STSMINB`, `STSMINH`.
  - `STUMAX`, `STUMAXB`, `STUMAXH`.
  - `STUMIN`, `STUMINB`, `STUMINH`.
- If FEAT_LRCPC is implemented, then `LDAPR, LDAPRB, LDAPRH`.
- If FEAT_LRCPC2 is implemented, then:
  - `LDAPURH, LDAPURSH, LDAPUR, LDAPURSW, LDAPURSB, LDAPURB`.
  - `STLUR, STLURH, STLURB`.

Move immediate to general-purpose register

These instructions are:

- `MOVZ`, `MOVN`, `MOVK` (immediate).
- `MOV` (between a general-purpose register and the stack pointer).

System instructions, Send Event, NOP, Clear Exclusive, and Prediction

In this context, the System instructions are the Cache maintenance instructions, TLB maintenance instructions, Address translation instructions, the prediction restriction instructions, and the check feature instruction.

These instructions are:

- `IC`.
- `DC`.
- `TLBI`.
- `AT`.
- `SEV`, `SEVL`.
- `NOP`.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- CLREX.
- CFP.
- CPP.
- DVP.

Pointer authentication instructions

These instructions are:

- If FEAT_PAuth is implemented, then:
  - AUTIA, AUTIAZ, AUTIZA.
  - AUTIB, AUTIBZ, AUTIZB.
  - AUTDA, AUTDZA.
  - AUTDB, AUTDZB.
  - PACIA, PACIAZ, PACIZA.
  - PACIB, PACIBZ, PACIZB.
  - PACDA, PACDZA.
  - PACDB, PACDZB.
  - PACGA.
  - XPACD, XPACI, XPACLRI.

E2.4.2.1.3 A64 instructions that are CONSTRAINED UNPREDICTABLE in Debug state

This subsection describes all instructions that are not listed in either:

- A64 instructions that are changed in Debug state.
- A64 instructions that are unchanged in Debug state.

These instructions are CONSTRAINED UNPREDICTABLE in Debug state. In general, the permissible behaviors are:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- If the instruction reads the PC or PSTATE, it uses an UNKNOWN value.
- If the instruction modifies the PC or PSTATE, other than by advancing the PC to the sequentially next instruction, it sets DLR_EL0 and DSPSR_EL0 to UNKNOWN values.
- If the instruction is similar to a Debug state instruction, it executes as that Debug state instruction.
- The instruction has the same behavior as in Non-debug state.

The following list shows the permissible behaviors for A64 instruction in Debug state. An instruction might appear multiple times in the list, in which case the choice of permissible behaviors is any of those listed. An example of this is CCMP.

Exception-generating instructions

These instructions are:

- SVC.
- HVC.
- SMC.
- BRK.
- HLT.

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a NOP.
- SVC behaves as DCPS1.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- `HVC` behaves as `DCPS2`.
- They generate the exception that the instruction would generate in Non-debug state. The exception is taken as described in Exceptions in Debug state.

Instructions that explicitly write to the PC (branches)

These instructions are:

- `B`, `B.cond`, `BL`, `BLR`, `BR`, `CBZ`, `CBNZ`, `RET`, `TBZ`, `TBNZ`.
- If FEAT_PAuth is implemented, then `RETAA`, `RETAB`, `BRAA`, `BRAB`, `BLRAA`, `BLRAB`, `BLRAAZ`, `BLRABZ`.

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a `NOP`.
- They execute as in Non-debug state without branching and set DSPSR_EL0 and DLR_EL0 to UNKNOWN values.

Exception return and related instructions

These instructions are:

- `ERET`.
- If FEAT_PAuth is implemented, then `ERETAA`, `ERETAB`.

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a `NOP`.
- They execute as in Non-debug state without branching, set DSPSR_EL0 and DLR_EL0 to UNKNOWN values, and either:
  - Execute the `DRPS` operation instead of performing an exception return, using UNKNOWN SPSR values.
  - Do not change the Exception level.

Instructions that request entry to a low-power state

These instructions are `WFE`, `WFI`.

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a `NOP`.
- They generate a synchronous exception if the corresponding instruction would be trapped in Non-debug state. See Configurable instruction controls.
- A `WFE` instruction clears the Event register if it is set.

Note

This means that these instructions must not suspend execution.

Instructions that read the PC

These instructions are:

- `LDR` (literal), `LDRSW` (literal).
- `ADR`, `ADRP`.
- `PRFM` (literal).

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a `NOP`.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- They execute as in Non-debug state, using an UNKNOWN value for the PC operand.

Instructions that explicitly modify PSTATE, other than `DCPS` and `DRPS`

These instructions are:

- `ADDS`, `SUBS`, `ADCS`, `SBCS`, `ANDS`, `BICS`, `CCMN`, `CCMP`.
- `FCMP`, `FCMPE`, `FCCMP`, `FCCMPE`.
- `MSR DAIFSet` (immediate), `MSR DAIFClr` (immediate), `MSR SPSel` (immediate).
- `MSR NZCV` (register), `MSR DAIF` (register), `MSR SPSel` (register).
- If FEAT_PAN is implemented, then:
  - `MSR PAN` (immediate).
  - `MSR PAN` (register).
- If FEAT_UAO is implemented, then:
  - `MSR UAO` (immediate).
  - `MSR UAO` (register).
- If FEAT_FlagM is implemented, then:
  - `CFINV`.
  - `RMIF`.
  - `SETF8`.
  - `SETF16`.
- If FEAT_DIT is implemented, then `MSR DIT`.

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a `NOP`.
- They execute as in Non-debug state, setting DSPSR_EL0 and DLR_EL0 to UNKNOWN values.
- For the `PACM` instruction, PSTATE.PACM is set to an UNKNOWN value.

Instructions that read PSTATE.{N, Z, C, V} or other PSTATE fields

These instructions are:

- `CSEL`, `CSINC`, `CSINV`, `CSNEG`, `CCMN`, `CCMP`, `FCSEL`, `FCCMP`, `FCCMPE`.
- `ADC`, `ADCS`, `SBC`, `SBCS`.
- `MRS NZCV`, `MRS DAIF`, `MRS SPSel`, `MRS CurrentEL`.
- If FEAT_PAN is implemented, then `MRS PAN`.
- If FEAT_UAO is implemented, then `MRS UAO`.
- If FEAT_FlagM is implemented, then `CFINV`.
- If FEAT_DIT is implemented, then `MRS DIT`.

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a `NOP`.
- They execute as in Non-debug state:
  - For the conditional operations and those that use Condition flags as an input, these instructions use UNKNOWN values for the Condition flag.
  - For the `MRS` instruction, they return an UNKNOWN value.
  - For the `RCW` and `RCWS` instructions, they return an UNKNOWN value.
  - For the `PACIASP` and `PACIBSP` instructions when PSTATE.PACM is 1, they use an UNKNOWN value for the PC operand.

Hint instructions

If FEAT_DGH is implemented, then this instruction is:

- `DGH`.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This instruction behaves in one of the following ways:

- It executes as a NOP.
- It executes as in Non-debug state.

All other instructions

These instructions behave in one of the following ways:

- They are UNDEFINED.
- They execute as a NOP.
- They execute as in Non-debug state.

Note

This includes instructions defined as UNPREDICTABLE or CONSTRAINED UNPREDICTABLE in Non-debug state. These instructions are UNPREDICTABLE or CONSTRAINED UNPREDICTABLE in Debug state.

E2.4.3 Decode tables

The syntax in the decode tables is defined as follows:

1	The bit has a fixed value of 1.
0	The bit has a fixed value of 0.
!=	The field has any value other than the value or values specified. The field might be an encoding field in the instruction whose value is supplied by the debugger.

Note

The instruction encodings in A64 Base Instruction Descriptions might show these bits as (0) or (1). A debugger must set these bits to 0 or 1, as appropriate.

Any other value indicates an encoding field in the instruction whose value is supplied by the debugger. Some values might be reserved or UNDEFINED, in which case the instruction is UNDEFINED or CONSTRAINED UNPREDICTABLE in Debug state, as it is in Non-debug state.

For more information about the instruction encodings, see A64 Base Instruction Descriptions.

For information about the syntax used in Figure E2-1, and see:

- Common syntax terms.
- Assembler symbols.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E2.4.3.1 A64 instructions that are modified in Debug state

Figure E2-1 shows the A64 instructions that are modified in Debug state. For information of how these are packed in the EDITR, see the register description.

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	Description
1 1 0 1	0 1 0 0	1 0 1	imm16			0	0 0 !=00	DCPS<opt>
1 1 0 1	0 1 0 1	0 0 L 1	1 0 1 1	0 1 0 0	0 1 0 1	0 0 0	Rt	MRS|MSR accessing DSPSR_EL0
1 1 0 1	0 1 0 1	0 0 L 1	1 0 1 1	0 1 0 0	0 1 0 1	0 0 1	Rt	MRS|MSR accessing DLR_EL0
1 1 0 1	0 1 1 0	1 0 1 1	1 1 1 1	0 0 0 0	0 0 1 1	1 1 1 0	0 0 0 0	DRPS
0 0 1 0	0 1 0 1	size 0	imm5	1 0 0	Pg	Zn	1	Pd	CMPNE (immediate)
1 1 0 0	0 0 0 0	0 1 0 0	1 1 1 0	0 off3	0 0 1 1	1 1 1	Rt	MOVT (scalar to table)
1 1 0 0	0 0 0 0	0 1 0 0	1 1 0 0	0 off3	0 0 1 1	1 1 1	Rt	MOVT (table to scalar)

Figure E2-1 Modified A64 instructions in Debug state

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E2.4.3.2 A64 instructions that are unchanged in Debug state

This section shows the A64 instructions that are unchanged in Debug state, other than some unallocated and UNDEFINED instructions.

31 30 29 28	27 26	25 24	23 22	21	20 19 18 17 16	15 14 13 12	11 10	9 8 7 6 5	4 3 2 1 0	Description
0 0 0 0	0 1	0 0	1 0	1	1 1 1 1 1	0 1 0 1	0	imm6	Rd	RDVL
0 0 0 0	0 1	0 0	1 0	1	1 1 1 1 1	0 1 0 1	1	imm6	Rd	RDSVL
0 0 0 0	0 1	0 1	0 0	1	imm8h	0 0 0	imm8l	Zm	Zdn	EXT
0 0 0 0	0 1	0 1	0 0	0 1	Pg	0	0 0	imm8	Zd	CPY (immediate, zeroing)
0 0 0 0	0 1	0 1	size	1	0 0 0 0 0	0 0 1 1	1 0	Rn	Zd	DUP (scalar)
0 0 0 0	0 1	0 1	size	1	0 0 1 0 0	0 0 1 1	1 0	Rm	Zdn	INSR (scalar)
sf 0 0	1 0 0	0 1	0 0	0	0 0 0 0 0	0 0 0 0	0 0	1 1 1 1 1	Rd	MOV <Rn>,SP
sf 0 0	1 0 0	0 1	0 0	0	0 0 0 0 0	0 0 0 0	0 0	Rn	1 1 1 1 1	MOV SP,<Rn>
0 0 0 1	1 0 0	0 1	A R	1	Rt2	0 0 0 1	0 0	Rn	Rt	LDCLRP
0 0 0 1	1 0 0	0 1	A R	1	Rt2	0 0 1 1	0 0	Rn	Rt	LDSETP
0 0 0 1	1 0 0	0 1	A R	1	Rt2	1 0 0 0	0 0	Rn	Rt	SWPP
0 0 1 0	0 1	0 1	0 0	0	1 1 0 0 1	1 1 1 1	0 0	0 0 0 0 0	Pd	RDFFR (unpredicated)
0 0 1	0 0	1 0	1 0	0	1 0 1 0 0	0 1 0 0	1 0	0 Pn	0 0 0 0 0	WRFFR
0 0 1 0	0 1	0 1	0 0	0	1 1 0 0	0 1 1 1	0 0	0 0 0 0 0	0 Pd	PTRUE
1 0 0	1 1 0	1 0	1 1	0	Rm	0 0 1 1	0 0	Rn	Rd	PACGA
sf !=01	1 0 0	1 0	1	hw	imm16				Rd	MOVN, MOVK, MOVZ
1 1 0 1	0 1 0 1	0	0	0	0 0 0 1 1	0 0 1 0	0	0 1 0 0 1	1 0 0 0 0	GCSB
1 1 0 1	0 1 0 1	0	0	0	0 0 0 1 1	0 0 1 0	0	0 0 0 0 0	0 1 1 1 1	NOP
1 1 0 1	0 1 0 1	0	0	0	0 0 0 1 1	0 0 1 0	0	0 1 0 1 1	0 1 1 1 1	CLRBHB
1 1 0	1 0 1	0 1	0 0	0	0 0 0 1 1	0 0 1 0	0	0 0 1 0	L 1 1 1 1	SEV, SEVL
1 1 0	1 0 1	0 1	0 0	0	0 0 0 1 1	0 0 1 0	0	0 0 1 1	1 1 1 1 1	XPACLRI
1 1 0	1 0 1	0 1	0 0	0	0 0 0 1 1	0 0 1 0	0	0 0	1 op2	1 1 1 1 1	When FEAT_PAuth_LR is not implemented, or PSTATE.PACM is 0: PAC(IA|IB)1716, AUT(IA|IB)1716
1 1 0	1 0 1	0 1	0 0	0	0 0 0 1 1	0 0 1 0	0	0 0	1 0 !=010	1 1 1 1 1	CSDB, ESB, PSB
1 1 0	1 0 1	0 1	0 0	0	0 0 0 1 1	0 0 1 0	0	0 0	1 0 0 1 0	1 1 1 1 1	TSB
1 1 0	1 0 1	0 1	0 0	0	0 0 0 1 1	0 0 1 0	0	0 0	1 1 op2	1 1 1 1 1	PAC(IA|IB)Z, AUT(IA|IB)Z When FEAT_PAuth_LR is not implemented, or PSTATE.PACM is 0: PAC(IA|IB)SP, AUT(IA|IB)SP
1 1 0 1	0 1 0 1	0	0	0	0 0 0 1 1	0 0 1 0	0	1 0 1 0 0	0 1 1 1 1	CHKFEAT
1 1 0	1 0 1	0 1	0 0	0	0 0 0 1 1	0 0 1 1	CRm	0 1 0	1 1 1 1 1	CLREX
1 1 0	1 0 1	0 1	0 0	0	0 0 0 1 1	0 0 1 1	option	1 op2	1 1 1 1 1	DSB, DMB, ISB, SB, SSBB, PSSBB
1 1 0 1	0 1 0 1	0	0	0	0 1 0 1 1	0 1 1 1	1 0	0 1 1 1 1	0 Rt	COSP
1 1 0 1	0 1 0 1	0	0	0	0 1 0 0 0	0 1 1 1	1 0	1 1 1 1 0	0 Rt	GCSPUSHX
1 1 0 1	0 1 0 1	0	0	0	0 1 0 0 0	0 1 1 1	1 0	1 1 1 1 1	0 Rt	GCSPOPX
1 1 0 1	0 1 0 1	0	0	0	0 1 0 1 1	0 1 1 1	1 0	1 1 1 0 1	0 Rt	GCSSS1
1 1 0 1	0 1 0 1	0	0	0	0 1 0 1 1	0 1 1 1	1 0	1 1 1 0 0	0 Rt	GCSPUSHM
1 1 0	1 0 1	0 1	0 0	0	0 1	op1	CRn	CRm	op2	Rt	IC, DC, TLBI, TLBIP, AT
1 1 0 1	1 0 0	1 0	0	0	1 1 1 1 1	0 0 0 0	1	1	Rn	Rt	GCSSTR
1 1 0 1	1 0 0	1 0	0	0	1 1 1 1 1	0 0 0 1	1	1	Rn	Rt	GCSSTTR
1 1 0 1	0 1 0 1	0	0	1	0 1 0 1 1	0 1 1 1	1 0	1 1 1 0 0	1 Rt	GCSPOPM

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Description
1	1	0	1	0	1	0	1	0	0	1	0	1	0	1	1	0	1	1	1	0	1	1	1	0	1	1	Rt					GCSSS2
1	1	0	1	0	1	0	1	0	0	L	1	0	op1			CRn				CRm				op2			Rt					MRS|MSR accessing System register
1	1	0	1	0	1	0	1	0	0	L	1	1	op1			!=0100				CRm				op2			Rt					MRS|MSR accessing System register
1	1	0	1	0	1	0	1	0	0	0	0	0	0	0	1	0	1	0	0	0	0	1	x	0	0	0	1	1	1	1	1	MSR PM, #<imm>
1	1	0	1	0	1	0	1	0	0	0	1	o0	op1			CRn				CRm				op2			Rt					MRS|MSR accessing PM
1	1	0	1	0	1	0	1	0	0	L	1	1	0	1	1	0	1	0	0	0	0	1	0	0	1	0	Rt					MRS|MSR accessing SVCR
1	1	0	1	0	1	0	1	0	0	L	1	1	op1			0	1	0	0	!=0010				op2			Rt					MRS|MSR accessing Special-purpose register
1	1	0	1	0	1	0	1	0	1	L	1	0	op1			CRn				CRm				op2			Rt					MRRS|MSRR
1	1	0	1	0	1	1	0	0	1	0	1	1	1	1	1	0	0	0	0	1	M	1	1	1	1	1	1	1	1	1	1	RETAA, RETAB
1	1	0	1	0	1	1	0	1	0	0	1	1	1	1	1	0	0	0	0	1	M	1	1	1	1	1	1	1	1	1	1	ERETAA, ERETAB
1	1	0	1	0	1	1	1	0	0	0	1	1	1	1	1	0	0	0	0	1	M	Rn					Rm					BRAA, BRAB
1	1	0	1	0	1	1	Z	0	0	1	1	1	1	1	1	0	0	0	0	1	M	Rn					Rm					BLRAA, BLRAB, BLRAAZ, BLRABZ
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	1	0	0	0	opc			Rn					Rd					PAC(IA|IB|DA|DB), AUT(IA|IB|DA|DB)
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	1	0	0	1	opc			1	1	1	1	1	Rd					PAC(IZA|IZB|DZA|DZB), AUT(IZA|IZB|DZA|DZB)
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	1	0	1	0	0	0	op	1	1	1	1	1	Rd					XPAC(I|D)
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	1	1	0	0	0	1	op	1	1	1	1	1	Rd					PACI(A|B)171615
1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	1	1	0	1	1	1	op	1	1	1	1	1	Rd					AUTI(A|B)171615
size		0	0	1	0	0	0	o2	L	0	Rs					o0	Rt2			Rn					Rt							LD(A|LA|X|AX)R{B|H}, ST(L|LL|X|LX)R{B|H}, CAS{A|L|AL}{B|H}
size		0	0	1	0	0	0	o2	L	1	Rs					o0	Rt2			Rn					Rt							LD{A}XP, ST{L}XP, CASP{A|L|AL}
0	x	0	1	1	0	0	1	opc		0	imm9									0	0	Rn			Rt							LDAPUR(B|H|SB|SH), STLUR(B|H)
1	0	0	1	1	0	0	1	!=11		0	imm9									0	0	Rn			Rt							LDAPUR{SW}, STLUR
1	1	0	1	1	0	0	1	!=1x		0	imm9									0	0	Rn			Rt							LDAPUR, STLUR
0	x	1	1	1	0	0	0	opc		0	imm9									0	0	Rn			Rt							LDUR(B|H|SB|SH),STUR(B|H)
1	0	1	1	1	0	0	0	!=11		0	imm9									0	0	Rn			Rt							LDUR{SW},STUR
1	1	1	1	1	0	0	0	!=1x		0	imm9									0	0	Rn			Rt							LDUR,STUR
size		1	1	1	0	0	0	opc		0	imm9									1	0	Rn			Rt							LDTR{B|H|SB|SH|SW}, STTR{B|H}
size		1	1	1	0	0	0	opc		0	imm9									P	1	Rn			Rt							LDR{B|H|SB|SH|SW}, STR{B|H}
size		1	1	1	0	0	0	A	R	1	Rs					0	opc			0	0	Rn			Rt							LD<op>{A|L|AL}{B|H}, ST<op>{A|L|AL}{B|H}
size		1	1	1	0	0	0	A	R	1	Rs					1	0	0	0	0	0	Rn			Rt							SWP{A|L|AL}{B|H}
size		1	1	1	0	0	0	1	0	1	Rs					1	1	0	0	0	0	Rn			Rt							LDAPR{B|H}
0	1	0	0	1	1	1	0	0	0	0	imm5					0	0	0	1	1	1	Rn			Rd							INS <Vd>.<Ts>[<index>],<R><n>
0	Q	0	0	1	1	1	0	0	0	0	imm5					0	0	1	1	1	1	Rn			Rd							UMOV <R><d>,<Vn>.<Ts>[<index>]
0	0	0	1	1	1	1	0	0	0	1	0	0	1	1	op	0	0	0	0	0	0	Rn			Rd							FMOV <Sd>,<Wn>, FMOV <Wd>,<Sn>
0	0	0	1	1	1	1	0	1	1	1	0	0	1	1	op	0	0	0	0	0	0	Rn			Rd							FMOV <Hd>, <Wn>, FMOV <Wd>, <Hn>

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Description
0	S	0	1	1	0	0	1	A	R	1	Rs					0	0	0	0	1	x	Rn					Rt					RCW{S}CAS{P}{A|L|AL}
0	S	0	1	1	0	0	1	A	R	1	Rt2					1	0	1	0	0	0	Rn					Rt					RCW{S}SWPP{A|L|AL}
0	S	0	1	1	0	0	1	A	R	1	Rt2					1	0	x	1	0	0	Rn					Rt					RCW{S}(CLRP|SETP){A|L|AL}
0	x	1	1	1	0	0	0	A	R	1	Rs					1	0	1	0	0	0	Rn					Rt					RCW{S}SWP{A|L|AL}
0	x	1	1	1	0	0	0	A	R	1	Rs					1	0	x	1	0	0	Rn					Rt					RCW{S}(CLR|SET){A|L|AL}
1	0	0	1	1	1	1	0	ft		1	1	0	0	1	1	op	0	0	0	0	0	0	0	Rn				Rd				FMOV <Dd|Hd>,<Xn>, FMOV <Xd>,<Dn|Hn>
1	0	0	1	1	1	1	0	1	0	1	0	1	1	1	op		0	0	0	0	0	0	0	Rn				Rd				FMOV <Vd>.D[1],<Xn> FMOV <Xd>,<Vn>.D[1]
1	0	0	1	0	0	0	1	1	0	uimm6					(0)	(0)	uimm4							Xn				Xd				ADDG
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	Q	V	Rs			Pg				Zn			0	off4				MOVA (vector to tile, single)
1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	Q	V	Rs			Pg			0	off4				Zd				MOVA (tile to vector, single)
1	1	0	1	1	0	0	1	0	0	1	imm9									1	0			Xn				Xd				STG - Signed offset
1	1	0	1	1	0	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0			Xn				Xd				STGM
1	1	0	1	1	0	0	1	1	0	1	imm9									1	0			Xn				Xd				ST2G - Signed offset
1	1	0	1	1	0	0	1	0	1	1	imm9									1	0			Xn				Xd				STZG - Signed offset
1	1	0	1	1	0	0	1	1	1	1	imm9									1	0			Xn				Xd				STZ2G - Signed offset
1	1	0	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0			Xn				Xd				STZGM
0	1	1	0	1	0	0	1	0	0	simm7						Xt2								Xn				Xt				STGP - Signed offset
1	1	0	1	0	0	0	1	1	0	uimm6						op3		uimm4						Xn				Xd				SUBG
1	1	0	1	1	0	0	1	0	1	1	imm9									0	0			Xn				Xd				LDG
1	1	0	1	1	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0			Xn				Xd				LDGM
1	1	1	1	1	0	0	0	0	0	1	1	1	1	1	1	1	0	0	1	0	0			Rn				Rt				ST64B
1	1	1	1	1	0	0	0	0	0	1	Rs					1	0	1	0	0	0			Rn				Rt				ST64BV0
1	1	1	1	1	0	0	0	0	0	1	Rs					1	0	1	1	0	0			Rn				Rt				ST64BV
1	1	1	1	1	0	0	0	0	0	1	1	1	1	1	1	1	1	0	1	0	0			Rn				Rt				LD64B

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E2.4.4 Security in Debug state

On entry to Debug state EDSCR.SDD is the Secure debug disabled flag. If entering Debug state from Secure state, EDSCR.SDD is set to 0.

Note

If halting is prohibited in a Security state, it is not possible to enter Debug state from that Security state. However, because changes to the authentication signals require a Context Synchronization event to guarantee their effect, there is a period during which the PE might halt even though the authentication signals prohibit halting.

In Debug state The value of EDSCR.SDD does not change, even if `ExternalSecureInvasiveDebugEnabled()` changes.

Note

- DBGAUTHSTATUS_EL1 is not frozen in Debug state. However, a Context Synchronization event is required to guarantee that changes are visible in DBGAUTHSTATUS_EL1.
- If EDSCR.SDD is 1 in Debug state, there is no means to enter a Security state where halting was prohibited on entry to Debug state. This means that it is not possible for the PE to be in such a Security state when EDSCR.SDD is 1. This is a general principle of behavior in Debug state.

In Non-debug state EDSCR.SDD returns the inverse of `ExternalSecureInvasiveDebugEnabled()`. If the authentication signals that control `ExternalSecureInvasiveDebugEnabled()` change, a Context Synchronization event is required to guarantee their effect.

Note

- A Context Synchronization event is also required to guarantee that changes in the authentication signals are visible in DBGAUTHSTATUS_EL1.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E2.4.5 Privilege in Debug state

The only additional privileges offered to Debug state are:

- The privilege to execute Debug state operations, DCPS, DRPS, MRS, MSR.
- The privilege to execute DTR access instructions regardless of the Exception level and traps.

The DTR access instructions can be executed at any Exception level, including EL0, regardless of any control register settings that might force these instructions to be UNDEFINED or trapped in Non-debug state. These instruction are:

- The MRS and MSR instructions that access DBGDTR_EL0, DBGDTRTX_EL0, and DBGDTRRX_EL0 in AArch64 state.

All other instructions operate with the privilege determined by the current Exception level and Security state. This applies to all Special-purpose and System registers accesses, memory accesses, and UNDEFINED instructions, and includes generating exceptions when the System registers trap or disable an instruction.

E2.4.6 Debug state operations, DCPS, DRPS, MRS, MSR

The architecture defines operations to change between Exception levels in Debug state. These operations can also change the mode at the current Exception level.

E2.4.6.1 DCPS<n>

Executing a DCPS<n> instruction in Debug state moves the PE to a higher Exception level.

The effect on endianness is the same as for exceptions and exception returns in Non-debug state. In AArch64 state the current endianness is determined by the value of SCTLR_ELx.EE for the target Exception level.

The DCPS<n> instructions are:

In AArch64 state

- DCPS1
- DCPS2
- DCPS3

The DCPS instructions are UNDEFINED in Non-debug state.

Table E2-2 shows the target of the instruction. In Table E2-2, the entries have the following meaning:

EL1h/Svc This means that the target is EL1h.

EL2h/Hyp This means that the target is EL2h.

Table E2-2 Target for DCPS instructions in Debug state

Instruction	Target when DCPS instruction executed at stated Exception level:		
	EL0	EL1	EL2
DCPS1	EL1h	EL1h	EL2h
DCPS2	EL2h	EL2h	EL2h
DCPS3	•	•	•

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Instruction	Target when DCPS instruction executed at stated Exception level:		
	EL0	EL1	EL2

However:

- DCPS1 is UNDEFINED at EL0 if EL2 is implemented and HCR_EL2.TGE == 1.
- DCPS3 is UNDEFINED at all Exception levels.

On executing a DCPS instruction:

- If the target Exception level is using AArch64:
  - ELR_ELx of the target Exception level becomes UNKNOWN.
  - SPSR_ELx of the target Exception level becomes UNKNOWN.
  - ESR_ELx of the target Exception level becomes UNKNOWN.
  - DLR_EL0 and DSPSR_EL0 become UNKNOWN.

The DCPSInstruction() function is described in A-profile Architecture Pseudocode.

E2.4.6.2 DRPS

Executing the DRPS operation in Debug state moves the PE to a lower Exception level, or the current Exception level, by copying the current SPSR to PSTATE.

If the target Exception level is the current Exception level, then the PE does not change Exception level.

The effect on endianness is the same as for exceptions and exception returns in Non-debug state.

Current endianness is set according to the Effective values of SCTLR_ELx.EE, or SCTLR_EL1.E0E for the target Exception level.

For more information see DRPS.

If the SPSR specifies an illegal exception return, then PSTATE.{nRW, EL, SP} are unchanged and PSTATE.IL is set to 1. For further information on illegal exception returns, see Illegal exception returns from AArch64 state.

Some PSTATE fields are ignored in Debug state. The effect of the DRPS operation on them is to set them to an UNKNOWN value that might be the value from the SPSR. For more information, see PSTATE in Debug state.

All other PSTATE fields are copied from SPSR.

DRPS is UNDEFINED at EL0 and in Non-debug state.

Note

Unlike an exception return, the DRPS operation has no architecturally-defined effect on the Event Register and Exclusives monitors. DRPS might set the Event Register or clear the Exclusives monitors, or both, but this is not a requirement and debuggers must not rely on any IMPLEMENTATION SPECIFIC behavior.

On executing a DRPS instruction, then DLR_EL0 and DSPSR_EL0 become UNKNOWN.

The DRPSInstruction() function is described in A-profile Architecture Pseudocode.

E2.4.6.3 MRS and MSR

The other Debug state instructions are used to read or write DLR_EL0 and DSPSR_EL0.

These instructions are:

In AArch64 state

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- MRS
- MSR (register)

```
MRS <Xt>, DLR_EL0          ; Copy DLR_EL0 to <Xt>
```

```
MRS <Xt>, DSPSR_EL0    ; Copy DSPSR_EL0 to <Xt>
```

```
MSR DLR_EL0, <Xt>      ; Copy <Xt> to DLR_EL0
```

```
MSR DSPSR_EL0, <Xt>    ; Copy <Xt> to DSPSR_EL0
```

These instructions can be executed at any Exception level when in Debug state, including EL0. They are UNDEFINED in Non-debug state.

E2.4.7 Exceptions in Debug state

The following sections describe how exceptions are handled in Debug state:

- Generating exceptions when in Debug state.
- Taking exceptions when in Debug state.
- Reset in Debug state.

E2.4.7.1 Generating exceptions when in Debug state

In Debug state:

- Instruction Abort exceptions cannot happen because instructions are not fetched from memory.
- Interrupts, including SError and virtual interrupts are ignored and remain pending:
  - The pending interrupt remains visible in ISR_EL1, Interrupt Status Register.
- Debug exceptions and debug events are ignored.
- Any attempt to execute an instruction bit pattern that is an allocated instruction at the current Exception level, but is listed in Executing instructions in Debug state as UNDEFINED in Debug state, generates an exception.
  The priority and syndrome for these exceptions is the same as for executing an encoding that does not have an allocated instruction.

Otherwise, synchronous exceptions are generated as they would be in Non-debug state and taken to the appropriate Exception level in Debug state.

E2.4.7.2 Taking exceptions when in Debug state

When the PE is in Debug state, all exceptions are synchronous. When an exception is generated, it is taken to Debug state. This means that:

- The target Exception level is as defined for the exception in Non-debug state.
- The exception syndrome is reported as defined for the exception in Non-debug state, except for the case described in Data Aborts in Memory access mode for which the reporting requirements are relaxed.
  The exception syndrome is reported using the syndrome register or registers for the target Exception level. In AArch64 state, these are ESR_ELx, and FAR_ELx. For example, if an instruction is trapped to an Exception level using AArch64 due to a configurable trap, disable, or enable, the exception code reported is the same as it would be in Non-debug state.

The effect on auxiliary syndrome registers, such as AFSR, is IMPLEMENTATION DEFINED.

- The PE remains in Debug state and changes to the target mode.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The endianness switches to that indicated by the EE bit of the System register for the target Exception level.
- The SPSR for the target Exception level or mode is corrupted and becomes UNKNOWN.
- If the target Exception level is using AArch64, ELR_ELx for the target Exception level becomes UNKNOWN.
- DLR_EL0, Debug Link Register and DSPSR_EL0, Debug Saved Program Status Register become UNKNOWN.
- The cumulative error flag, EDSCR.ERR, is set to 1. See Cumulative error flag.
- PSTATE.IL is cleared to 0.
- PSTATE.{IT, T, SS, D, A, I, F} are set to UNKNOWN values, and PSTATE.{N, Z, C, V, Q, GE} are unchanged. However, these fields are ignored and are not observable in Debug state. For more information, see PSTATE in Debug state.

The debugger must save any state that can be corrupted by an exception before executing an instruction that might generate another exception.

E2.4.7.3 Pseudocode description of taking exceptions in Debug state

`AArch64.TakeException()` shows the behavior when the PE takes an exception to an Exception level using AArch64 in Non-debug state. In Debug state, this redirects to `AArch64.TakeExceptionInDebugState()`.

E2.4.7.4 Reset in Debug state

If the PE is reset when in Debug state, it exits Debug state and enters Non-debug reset state. When the PE is in reset state, EDSCR.STATUS == `0b000010` and writes to EDITR are ignored.

Note

If EDECR.RCE == 1 or CTIDEVCTL.RCE ==1, meaning that a Reset Catch debug event is programmed, and if halting is allowed on exiting reset state, then on exiting reset state the PE halts and re-enters Debug state. See Reset Catch debug events. All PE registers have taken their reset values, which might be UNKNOWN.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E2.4.8 Accessing registers in Debug state

Register accesses are unchanged in Debug state. The view of each register is determined by either the current Exception level or the mode, or both, and accesses might be disabled or trapped by controls at a higher Exception level.

E2.4.8.1 General-purpose register access, other than AArch64 state SP access

A single general-purpose register can be read by issuing an `MSR` instruction through the ITR to write DBGDTR_EL0 in AArch64 state. The debugger can then read the DTR register or registers through the external debug interface. The reverse sequence writes to a general-purpose register.

Figure E2-2 shows the reading and writing of general-purpose registers, other than SP, in Debug state in AArch64 state.

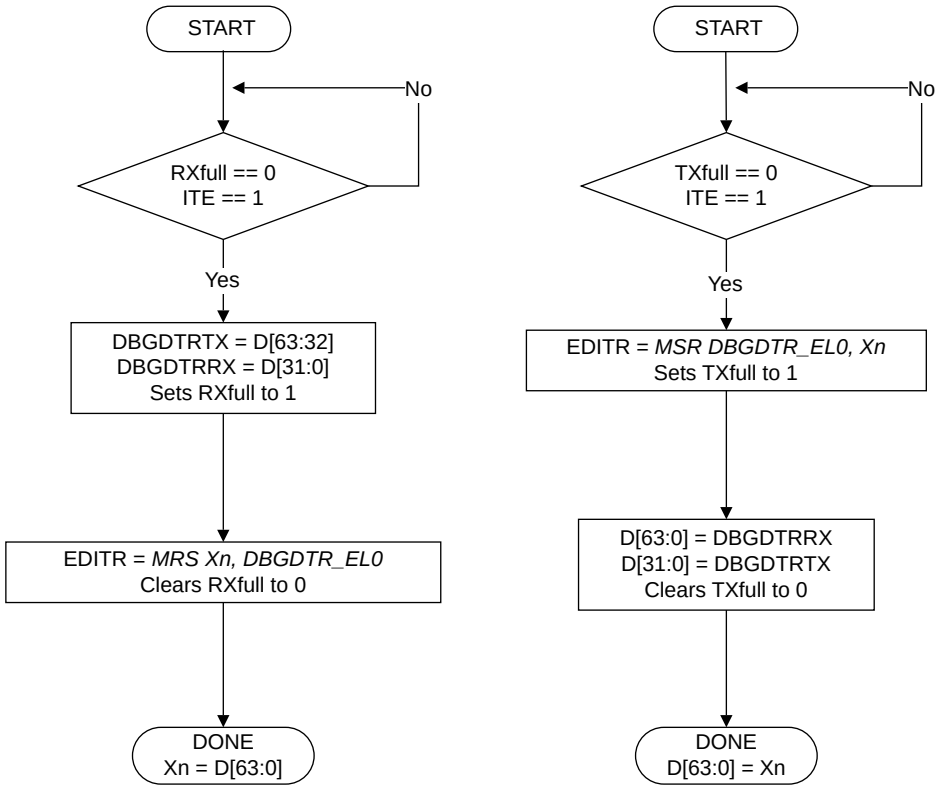


Figure E2-2 Reading and writing general-purpose registers, other than SP, in Debug state in AArch64 state

E2.4.8.2 SIMD&FP register, System register, and AArch64 state SP accesses

To read a SIMD&FP register or System register, the debugger must first copy the value into a general-purpose register using:

- An `FMOV` instruction for floating-point transfers to SIMD&FP registers.
- A `UMOV` instruction for SIMD transfers to SIMD&FP registers.
- An `MRS` instruction for System registers.
- A `MOV Xd,SP` instruction for the SP register.

The debugger can then read out the particular general-purpose register. The reverse sequence writes a register.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E2.4.8.3 PC and PSTATE access

The debugger reads the Program Counter and PSTATE of the process being debugged through the DLR_EL0 and DSPSR_EL0 System registers. The actual values of PC and PSTATE cannot be directly observed in Debug state:

- Instructions that are used for direct reads and writes of PC and PSTATE in Non-debug state are CONSTRAINED UNPREDICTABLE in Debug state.
- On taking an exception, ELR_ELx and SPSR_ELx at the target Exception level are UNKNOWN. They do not record the PC and PSTATE.

PSTATE.{UAO, PAN, IL, nRW, EL, M} are indirectly read by instructions executed in Debug state, but all other PSTATE fields are ignored and cannot be observed. See also:

- PSTATE in Debug state.
- Executing instructions in Debug state.
- Exceptions in Debug state.

E2.4.9 Accessing memory in Debug state

How the PE accesses memory is unchanged in Debug state. This includes:

- The operation of the MMU, including address translation, access permissions, memory attribute determination, and the operation of any TLBs.
- The operation of any caches and coherency mechanisms.
- Alignment support.
- Endianness support.
- The Memory order model.
- The operation of the MPU, including accesse permissions and memory attribute determination.

E2.4.9.1 Simple memory transfers

Simple memory accesses can be performed in Debug state by issuing memory access instructions through the ITR and passing data through the DTR registers. Executing instructions in Debug state lists the memory access instructions that are supported in Debug state.

E2.4.9.2 Bulk memory transfers

Memory access mode can accelerate bulk memory transfers in Debug state. See DCC and ITR access modes.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E2.5 Exiting Debug state

The PE exits Debug state when it receives a Restart request trigger event. If EDSCR.ITE == 0, when the PE receives a Restart Request trigger event, the behavior of any instruction issued through the ITR in Normal access mode or an operation issued by a DTR access in Memory access mode that has not completed execution is one of the following:

- The instruction completes execution in Debug state and the PE ignores the restart request and remains in Debug state.
- The instruction completes execution in Debug state before the PE executes the restart sequence.
- The instruction completes execution in Non-debug state after the PE executes the restart sequence.
- The instruction is abandoned. This means that the instruction does not execute. Any registers or memory accessed by the instruction are left in an UNKNOWN state.

Note

- Implementations can set EDSCR.ITE to 1 to indicate that further instructions can be accepted by ITR before the previous instructions have completed. If any previous instruction has not completed and EDSCR.ITE == 1, then the PE must complete these instructions in Debug state before executing the restart sequence. EDSCR.ITE == 0 indicates that the PE is not ready to restart.
- A debugger must observe that any instructions issued through EDITR that might generate a synchronous exception, as complete, before issuing a restart request. It can do this by observing the completion of a later instruction, as synchronous exceptions must occur in program order. For example, a debugger can observe that an instruction that reads or writes a DTR register is complete because of its effect on the EDSCR.{TXfull, RXfull} flags.

On exiting Debug state, the PE sets the Program Counter to the address in DLR_EL0, Debug Link Register, where:

- If exiting to AArch64 state:
  - Bits[63:56] of DLR_EL0 might be ignored as part of tagged address handling. See Address tagging.
  - Otherwise the PC is set from DLR_EL0.

Exit from Debug state can give rise to a PC alignment fault exception when the Program Counter is used. For more information, see PC alignment checking.

On exiting Debug state, PSTATE is set from DSPSR_EL0, Debug Saved Program Status Register in the same way that an exception return sets PSTATE from SPSR_ELx:

- The same illegal exception return checks that apply to an exception return also apply to exiting Debug state. If the return from Debug state is an illegal exception return then the effect on PSTATE and the PC is the same as for any other illegal exception return. See Exception return.
- PSTATE.SS is copied from DSPSR_EL0, Debug Saved Program Status Register.SS if all of the following hold:
  - MDSCR_EL1.SS == 1.
  - The debug target Exception level is using AArch64.
  - Software step exceptions from the restart Exception level are enabled.

  Otherwise PSTATE.SS is set to 0.

  Note

  Unlike a return using ERET, PSTATE.SS must be restored from DSPSR_EL0, Debug Saved Program Status Register.SS because otherwise it is UNKNOWN.

  However, if OSDLR_EL1, OS Double Lock Register.DLK == 1 and DBGPRCR_EL1, Debug Power Control Register.CORENPDRQ == 0, meaning FEAT_DoubleLock is implemented and locked in Non-debug state and therefore Software Step exceptions are disabled, but otherwise Software Step exceptions would be enabled from the restart Exception level, it is CONSTRAINED UNPREDICTABLE whether PSTATE.SS is copied from DSPSR_EL0, Debug Saved Program Status Register.SS.
- If FEAT_SSBS is implemented, DSPSR_EL0, Debug Saved Program Status Register.SSBS is copied to PSTATE.SSBS.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If FEAT_PAN is implemented, DSPSR_EL0, Debug Saved Program Status Register.PAN is copied to PSTATE.PAN.
- If FEAT_UAO is implemented, on exit from Debug state, DSPSR_EL0.UAO is copied to PSTATE.UAO.
- If FEAT_DIT is implemented, on exit from Debug state, DSPSR_EL0, Debug Saved Program Status Register.DIT is copied to PSTATE.DIT.

Note

- One important difference between Debug state exit and an exception return is that the PE can exit Debug state at EL0. Despite this, the behavior of an exit from Debug state is similar to an exception return. For example, PSTATE.{D, A, I, F} is updated regardless of the value of SCTLR_EL1.UMA.
- Exit from Debug state has no architecturally-defined effect on the Event Register and Exclusives monitors. An exit from Debug state might set the Event Register or clear the Exclusives monitors, or both, but this is not a requirement and debuggers must not rely on any IMPLEMENTATION SPECIFIC behavior.

The `ExitDebugState()` function is described in A-profile Architecture Pseudocode.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter E3
Halting Debug Events

This chapter describes a particular class of debug events. It contains the following sections:

- Introduction to Halting debug events.
- Halting Step debug events.
- Halt Instruction debug event.
- Exception Catch debug event.
- External Debug Request debug event.
- OS Unlock Catch debug event.
- Reset Catch debug events.
- Software Access debug event.
- Synchronization and Halting debug events.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E3.1 Introduction to Halting debug events

External debug defines Halting debug events. The following Halting debug events are available from the introduction of Armv8:

- Halting Step debug events.
- Halt Instruction debug event.
- Exception Catch debug event.
- External Debug Request debug event.
- OS Unlock Catch debug event.
- Reset Catch debug events.
- Software Access debug event.

If halting is allowed, a Halting debug event halts the PE. The PE enters Debug state.

In addition, breakpoints and watchpoints might halt the PE if halting is allowed. See Breakpoint and Watchpoint debug events. Because breakpoints and watchpoints can generate an exception or halt the PE, Breakpoint and Watchpoint debug events are not classified as Halting debug events.

For a definition of Debug state, see Debug State. For a definition of halting allowed, see Halting allowed and halting prohibited.

Debug state entry and debug event prioritization describes the behavior when multiple debug events are generated by an instruction.

See also Synchronization and Halting debug events.

Table E3-1 shows the behavior of Breakpoint, Watchpoint, and Halting debug events.

Table E3-1 Summary of debug events and possible outcomes

Debug event type	PE behavior when halting is:	
	Allowed	Prohibited
Breakpoint and Watchpoint debug events	Halt	See Table D2-1
Halt Instruction debug event	Halt	UNDEFINED
Software Access debug event	Halt	Ignored
Exception Catch debug event	Halt	Ignored or pended
Halting Step debug events	Halt	Pended
External Debug Request debug event	Halt	Pended
Reset Catch debug events	Halt	Pended
OS Unlock Catch debug event	Pended	Pended

Table E3-2 shows where the pseudocode for each Halting debug event type is located.

Table E3-2 Pseudocode description of Halting debug events

Halting debug event type	Pseudocode
Halt Instruction debug event	HLT
Software Access debug event	Pseudocode description of Software Access debug event

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Halting debug event type	Pseudocode
Exception Catch debug event	Pseudocode description of Exception Catch debug events
Halting Step debug events	Pseudocode description of Halting Step debug events
External Debug Request debug event	Pseudocode description of External Debug Request debug events
Reset Catch debug events	Pseudocode description of Reset Catch debug event
OS Unlock Catch debug event	Pseudocode description of OS Unlock Catch debug event

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E3.2 Halting Step debug events

Halting Step is a debug resource that a debugger can use to make the PE step through code one instruction at a time. This section describes the Halting Step debug events. It is divided into the following sections:

- Overview of a Halting Step debug event.
- The Halting Step state machine.
- Using Halting Step.
- Detailed Halting Step state machine behavior.
- Synchronization and the Halting Step state machine.
- Disabling interrupts while stepping.
- Syndrome information on Halting Step.
- Pseudocode description of Halting Step debug events.

The architecture describes the behavior as a simple Halting Step state machine. See The Halting Step state machine.

E3.2.1 Overview of a Halting Step debug event

The behavior of Halting Step is defined by a state machine, shown in Figure E3-1. A Halting Step debug event executes a single instruction and then returns control to the debugger. When the debugger software wants to execute a Halting Step:

1. With the PE in Debug state, the debugger activates Halting Step.
2. The debugger signals the PE to exit Debug state and return to the instruction that is to be stepped.
3. The PE executes that single instruction.
4. The PE enters Debug state before executing the next instruction.

However, an exception might be generated while the instruction is being stepped. That is either:

- A synchronous exception generated by the instruction being stepped.
- An asynchronous exception taken before or after the instruction being stepped.

Halting Step has its own enable control bit, EDECR.SS and EDESR.SS.

Note

Because the Halting Step state machine states occur as a result of normal PE operation, the states can be described as both:

- PE states.
- Halting Step states.

E3.2.2 The Halting Step state machine

The state machine states are:

Inactive Halting Step is inactive. No Halting Step debug events can be generated, therefore execution is not affected by Halting Step. The PE is in this state whenever either of the following is true:

- Halting Step is disabled. That is, EDECR.SS is set to 0 and EDESR.SS is set to 0.
- Halting is prohibited. See Halting the PE on debug events. In this state, if EDECR.SS is set to 1, then a Halting Step debug event is pending.

In Figure E3-1, this state is shown in red.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Active-not-pending

Halting Step is enabled and active. This is the state in which the PE steps an instruction. EDECR.SS == 1 and EDESR.SS == 0. Software must not set EDECR.SS to 1 unless the PE is in Debug state, otherwise behavior is CONSTRAINED UNPREDICTABLE, as described in Changing the value of EDECR.SS when not in Debug state.

In Figure E3-1, this state is shown in green.

Active-pending

Halting Step is enabled and active. The step has completed and the PE enters Debug state. EDESR.SS == 1.

In Figure E3-1, this state is shown in green.

Whenever Halting Step is enabled and active, whether the state machine is in the active-not-pending state or in the active-pending state depends on EDESR.SS. Table E3-3 shows this.

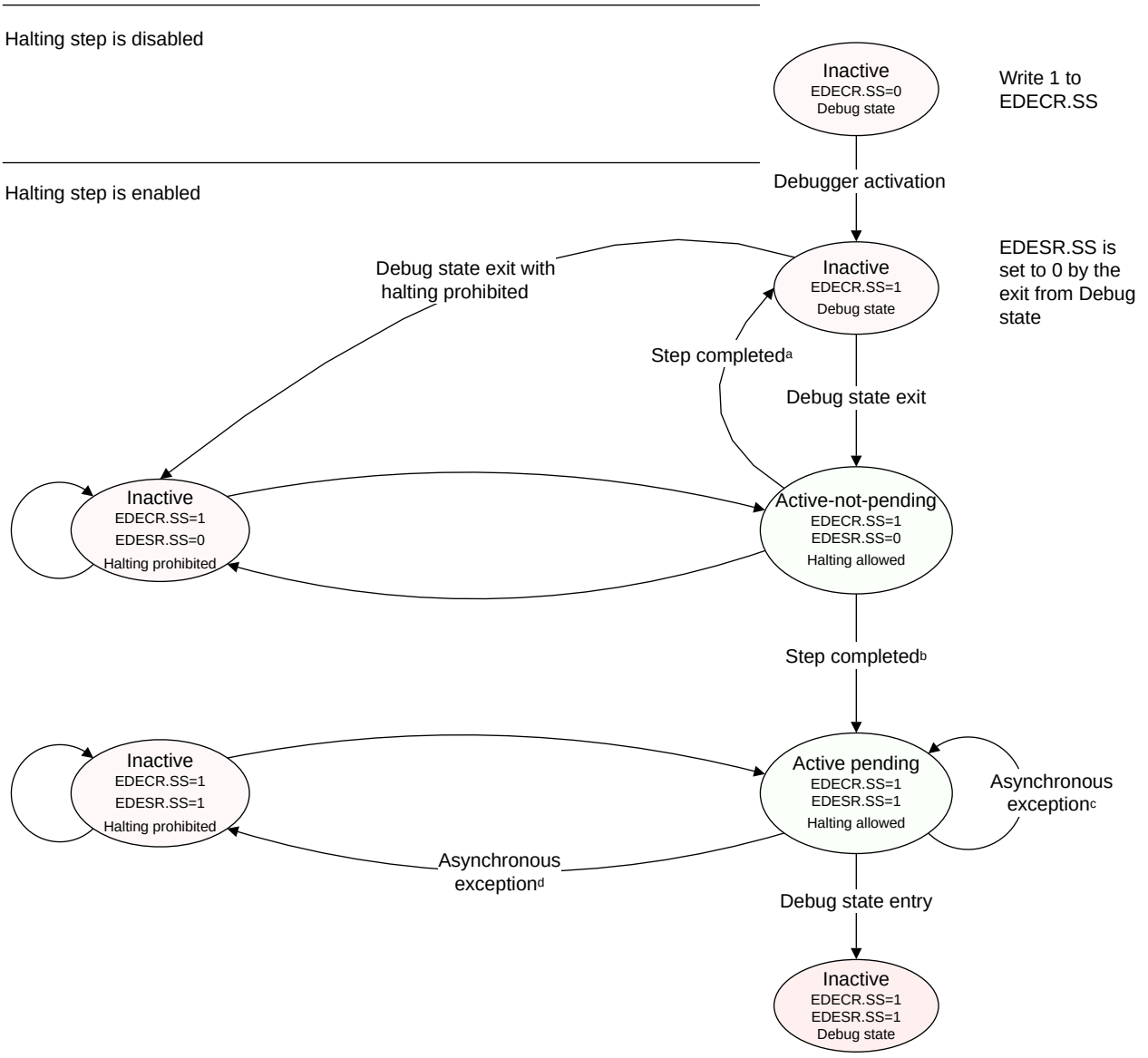
In the simple sequential execution of the program, the PE executes the Halting Step state machine as follows:

1. Initially, Halting Step is inactive.
2. After exiting Debug state, Halting Step is active-not-pending.
3. The PE executes an instruction and Halting Step is active-pending.
4. The pending Debug state entry is taken on the next instruction and the step is complete.

Exceptions and other changes to the PE context can interrupt this sequence.

Figure E3-1 shows a Halting Step state machine.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential



a. Step completed occurs when:
   - A debug event, other than a Halting Step debug event, causes entry into Debug state.

b. Step completed occurs when:
   - An instruction is executed without taking an exception.
   - An exception is taken to a state where halting is allowed.
   - A reset.

c. An asynchronous exception taken to a state where halting is allowed.

d. An asynchronous exception taken to Secure state where halting is prohibited.

Figure E3-1 Halting Step state machine

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Figure E3-1 describes only state transitions to and from the inactive state by exit from Debug state, executing an exception return, or taking an exception. Other changes to the PE context, including writes to registers such as EDECR and OSDLR_EL1, OS Double Lock Register and changes to the authentication interface can also cause changes to the Halting Step state machine. These can lead to UNPREDICTABLE or CONSTRAINED UNPREDICTABLE behavior. See Synchronization and the Halting Step state machine.

The following bits control the state machine, as shown in Table E3-3:

- EDECR.SS. This is the Halting Step enable bit.
- EDESR.SS. This is the Halting Step debug event pending bit.

Table E3-3 shows the Halting Step state machine states. The letter X in a register column means that the relevant bit can be set to either zero or one.

Table E3-3 Halting Step state machine states

Halting	EDECR.SS	EDESR.SS	Halting Step state
Prohibited	X	X	Inactive (Halting Step debug even not pending)
Prohibited	X	1	Inactive (Halting Step debug event pending)
Allowed	0	0	Inactive
Allowed	1	0	Active-not-pending
Allowed	X	1	Active-pending

E3.2.3 Using Halting Step

To step a single instruction the PE must be in Debug state:

1. The debugger sets EDECR.SS to 1 to enable Halting step.
2. The debugger signals the PE to exit Debug state with DLR_EL0, Debug Link Register set to the address of the instruction being stepped. The PE clears EDESR.SS to 0 and the Halting Step state machine enters the active-not-pending state.
3. The PE executes the instruction being stepped.
   If an exception is taken to a state where halting is prohibited, then EDESR.SS is always correct for the preferred return address of the exception.
4. The PE enters Debug state before executing the next instruction and the step is complete.

Note

- If FEAT_DoPD is not implemented, EDECR.SS value is in the Debug power domain, meaning that the state machine is maintained over a powerdown of the Core power domain.
- If FEAT_DoPD is implemented, the values of EDECR.SS and EDESR.SS are set to 0 on a Cold reset, and, if the PE was stepping an instruction, EDESR.SS is effectively UNKNOWN after a Warm reset. A debugger must use a Reset Catch debug event to step over a powerdown state.
- A debugger must only change the value of EDECR.SS when the PE is in Debug state, otherwise behavior is CONSTRAINED UNPREDICTABLE as described in Changing the value of EDECR.SS when not in Debug state.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E3.2.4 Detailed Halting Step state machine behavior

The behavior of the Halting Step state machine is described in the following sections:

- Entering the active-not-pending state.
- PE behavior in the active-not-pending state.
- Entering the active-pending state.
- PE behavior in the active-pending state.
- PE behavior in the inactive state when in Non-debug state.
- PE behavior in Debug state.

E3.2.4.1 Entering the active-not-pending state

The PE enters the active-not-pending state:

- By exiting Debug state to a state where halting is allowed with EDECR.SS == 1.
- By an exception return from a state where halting is prohibited to a state where halting is allowed with EDECR.SS == 1 and EDESR.SS == 0.
- As described in Synchronization and the Halting Step state machine.

E3.2.4.2 PE behavior in the active-not-pending state

When the PE is in the active-not-pending state it does one of the following:

- It executes one instruction and does one of the following:
  - Completes it without taking a synchronous exception.
  - Takes a synchronous exception generated by the instruction.
  - Generates a debug event that causes entry to Debug state.
- It takes an asynchronous exception without executing any instruction.
- It takes an asynchronous debug event into Debug state.

E3.2.4.2.1 If no exception or debug event is generated

If no exception or debug event is generated the PE sets EDESR.SS to 1. This means that the Halting Step state machine advances to the active-pending state.

E3.2.4.2.2 If an exception or debug event is generated

The PE sets EDESR.SS according to all of the following:

- The type of exception.
- The target Exception level of the exception.

If an exception or debug event is generated, the PE sets EDESR.SS to 1 if the exception is an SMC exception or is taken to a state where halting is allowed.

Otherwise, EDESR.SS is unchanged.

It is UNPREDICTABLE whether EDESR.SS is set to 1 or unchanged when an SError exception is taken to EL3 without executing the instruction, and halting is prohibited at EL3.

If halting is prohibited after taking the exception or debug event, then the Halting Step state machine advances to the inactive state. Otherwise, the Halting Step state machine advances to the active-pending state.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

The underlying criteria for the value of EDESR.SS on an exception are:

- Whether halting is allowed at the target of the exception. If halting is allowed, the PE must step into the exception. If halting is prohibited, the PE must step over the exception.
- Whether the preferred return address of the exception is the instruction itself or the next instruction, if the PE steps over the exception.

Table E3-4 shows the behavior of the active-not-pending state.

Table E3-4 Summary of active-not-pending state behavior

Event	Target Exception level	Value written to EDESR.SS
No exception or debug event	Not applicable	1
SMC exception	EL2 if trapped	1
Reset	Highest	1
Exception, other than SMC exception	EL1	1
	EL2	1
Debug event	Debug state	Unchanged

E3.2.4.3 Entering the active-pending state

The PE enters the active-pending state by one of the following:

- From the active-not-pending state by:
  - Executing an instruction without taking an exception.
  - Taking an exception so that the PE remains in a state where halting is allowed.
- An exception return from a state where halting is prohibited when EDESR.SS == 1.
- A reset when the value of EDECR.SS == 1, regardless of the state the PE was in before the reset occurred.
- From the active-pending state by taking an asynchronous exception to a state where halting is allowed.
- Following the description in Synchronization and the Halting Step state machine.

E3.2.4.4 PE behavior in the active-pending state

When the PE is in the active-pending state, it enters Debug state before executing an instruction.

The entry into Debug state has higher priority than all other types of synchronous debug event and synchronous exception. However, the architecture does not define the prioritization of this Debug state entry with respect to any unmasked pending asynchronous exception. If an asynchronous exception is prioritized over the entry to Debug state, then EDESR.SS is unchanged.

For more information on the prioritization of debug events, see Debug state entry and debug event prioritization.

E3.2.4.5 PE behavior in the inactive state when in Non-debug state

EDESR.SS is not updated by the execution of an instruction or the taking of an exception when Halting Step is inactive. This means that EDESR.SS is not changed by an exception handled in a state where halting is prohibited.

On return to a state where halting is allowed, the Halting Step state machine is restored either to the active-pending state or the active-not-pending state, depending on the value of EDESR.SS. The return to a state where halting is allowed is normally by an exception return, which in some situations is a Context Synchronization event.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

See also Synchronization and the Halting Step state machine.

E3.2.4.6 PE behavior in Debug state

Halting Step is inactive in Debug state because halting is prohibited, see Halting allowed and halting prohibited.

Entry to Debug state does not change EDESR.SS.

EDESR.SS is cleared to 0 on exiting Debug state as the result of a restart request. If EDECR.SS == 1, Halting Step enters the active-not-pending state.

Note

This means that EDESR.SS is never cleared to 0 by the execution of an instruction in Debug state, or by taking an exception when in Debug state as described in PE behavior in the active-not-pending state, because the Halting Step state machine is not in the active-not-pending state. EDESR.SS can be cleared by a write to EDESR, see the register description.

However, if the PE exits Debug state as the result of a PE reset and EDECR.SS == 1, then Halting Step immediately enters the active-pending state, as EDESR.SS is set to the value of EDECR.SS.

E3.2.5 Synchronization and the Halting Step state machine

The Halting Step state machine also changes state if:

- Halting becomes allowed or prohibited other than by exit from Debug state, an exception return, or taking an exception. This means that halting becomes allowed or prohibited because:
  - The Security state changes without an exception return. See State and mode changes without explicit Context Synchronization events.
  - The external authentication interface changes.
  - FEAT_DoubleLock is implemented and the status, `DoubleLockStatus()`, changes.
- A write to EDECR when the PE is in Non-debug state changes the value of EDECR.SS.

  Note

  Behavior is CONSTRAINED UNPREDICTABLE if the value of EDECR.SS is changed when the PE is in Non-debug state, see Changing the value of EDECR.SS when not in Debug state.

- A write to EDESR when the PE is in Non-debug state clears EDESR.SS to 0.

These operations are guaranteed to take effect only after a Context Synchronization event. If the instruction being stepped generates a Context Synchronization event, then the PE might use the old or new state.

The PE must perform the required behavior of the new state before or immediately following the next Context Synchronization event, but it is not required to do so immediately. This means that the PE can perform the required behavior of the old state before the next Context Synchronization event. This is illustrated in Example E3-1 and Example E3-2.

Example E3-1 Synchronization requirements 1

EDECR.SS is set to 1 in Debug state, requesting the active-not-pending state on exit from Debug state. On exit from Debug state the PE immediately takes an exception to EL2, where halting is prohibited. The PE does not step any instructions but executes the software at EL2 as normal. EDESR.SS remains set to 0. If halting is subsequently allowed, the PE must perform the required behavior of the active-not-pending state before or immediately following the next Context Synchronization event, but it is not required to do so immediately.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example E3-2 Synchronization requirements 2

EDECR.SS is set to 1 in Debug state. On exit from Debug the PE executes an `MSR` instruction that sets OSDLR_EL1.DLK to 1 and `DoubleLockStatus()` becomes TRUE. This change requires a Context Synchronization event to guarantee its effect, meaning it is CONSTRAINED UNPREDICTABLE whether:

- Halting is allowed:
  - The PE enters Debug state on the next instruction.
- Halting is prohibited:
  - The PE does not enter Debug state.

The value in EDESR.SS depends on whether halting was allowed or prohibited when the write to OSDLR_EL1.DLK completed, and so it might be 0 or 1. If a second `MSR` instruction clears OSDLR_EL1.DLK to 0, the PE must perform the required behavior of the state indicated by EDESR.SS before or immediately following the next Context Synchronization event, but it is not required to do so immediately.

See also Synchronization and Halting debug events.

E3.2.5.1 Changing the value of EDECR.SS when not in Debug state

If software changes the value of EDECR.SS when the PE is not in Debug state then behavior is CONSTRAINED UNPREDICTABLE, and one or more of the following behaviors occurs:

- The value of EDECR.SS becomes UNKNOWN.
- The state of the Halting Step state machine becomes UNKNOWN.
- On a reset of the PE, the value of EDECR.SS and the state of the Halting Step state machine are UNKNOWN.

E3.2.6 Disabling interrupts while stepping

When using Halting Step, the sequence of entering Debug state, interacting with the debugger, and then exiting Debug state for each instruction reduces the rate at which the PE executes instructions. However, the rate at which certain interrupts, such as timer interrupts, are generated might be fixed by the system. This means it might be necessary to disable interrupts while using Halting Step by setting EDSCR.INTdis, to allow the code being debugged to make forward progress.

E3.2.7 Syndrome information on Halting Step

Three EDSCR.STATUS encodings record different scenarios for entering Debug state on a Halting Step debug event:

Halting Step, normal

An instruction other than a Load-Exclusive instruction was stepped.

Halting Step, exclusive

A Load-Exclusive instruction was stepped.

Halting Step, no syndrome

The syndrome data is not available.

If the PE enters Debug state due to a Halting Step debug event immediately after stepping an instruction in the active-not-pending state, EDSCR.STATUS is set to either:

- Halting Step, normal, if the stepped instruction was not a Load-Exclusive instruction.
- Halting Step, exclusive, if the stepped instruction was a Load-Exclusive instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If the stepped instruction was a conditional Load-Exclusive instruction that failed its Condition code check, EDSCR.STATUS is set to a CONSTRAINED UNPREDICTABLE choice of Halting Step, normal, or Halting Step, exclusive.

Otherwise, the PE enters Debug state without stepping an instruction. This means that the Halting Step state machine enters the active-pending state directly from the inactive state, without going through active-not-pending state. In this case, EDSCR.STATUS is set to Halting Step, no syndrome.

In addition, EDSCR.STATUS is CONSTRAINED UNPREDICTABLE when:

- A different exception is taken before the Halting Step debug event.
  In this case EDSCR.STATUS is set to a CONSTRAINED UNPREDICTABLE choice of:
  - Halting Step, no syndrome, or Halting Step, normal, if the stepped instruction was not a Load-Exclusive instruction.
  - Halting Step, no syndrome, or Halting Step, exclusive, if the stepped instruction was a Load-Exclusive instruction.
- The instruction that was stepped was an Exception Return instruction or an `ISB`. As these instructions are not in the Load-Exclusive instructions, EDSCR.STATUS is set to a CONSTRAINED UNPREDICTABLE choice of Halting Step, no syndrome or Halting Step, normal.
- The PE enters directly into the active-pending state on a Warm reset because EDECR.SS is set to 1. EDSCR.STATUS is set to a CONSTRAINED UNPREDICTABLE choice of Halting Step, no syndrome or Halting Step, normal.

In all cases, if EDSCR.STATUS is not set to Halting Step, no syndrome, then it must indicate whether the stepped instruction was a Load-Exclusive instruction by setting EDSCR.STATUS to Halting Step, normal or Halting Step, exclusive.

Note

If the PE cannot determine whether the stepped instruction was a Load-Exclusive instruction or not, then it sets EDSCR.STATUS to Halting Step, no syndrome. For example, the exception is taken before the PE decodes the stepped instruction, or the exception means the PE has no valid stepped instruction to decode.

Note

The architecture does not permit an implementation to always set EDSCR.STATUS to Halting Step, no syndrome.

E3.2.8 Pseudocode description of Halting Step debug events

There are two pseudocode functions for Halting Step debug events:

- `HSAdvance()`. This is called after an instruction has executed and any exception generated by the instruction is taken, except for the case of an SVC, HVC, or SMC exception, in which case it is called before the exception is taken. `HSAdvance()` affects the next instruction.
- `CheckHaltingStep()`. This is called before the next instruction is executed. If a step is pending, it generates the debug event.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E3.3 Halt Instruction debug event

A Halt Instruction debug event is generated when EDSCR.HDE == 1, halting is allowed, and the PE executes the Halt instruction, `HLT`.

The pseudocode for Halt Instruction debug events is described in HLT.

`HLT` never generates a debug exception. It is treated as UNDEFINED if EDSCR.HDE == 0, or if halting is prohibited.

Note

A debugger can replace a program instruction with a Halt instruction to generate a Halt Instruction debug event. Debuggers that use the `HLT` instruction must be aware of the rules for concurrent modification of executable code, CMODX. The rules for concurrent modification and execution of instructions do not allow one thread of execution or an external debugger to replace an instruction with an `HLT` instruction when these same instructions are being executed by a different thread of execution. See Concurrent modification and execution of instructions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E3.4 Exception Catch debug event

Exception Catch debug events:

- Are generated when the corresponding bit in the Exception Catch Control Register, EDECCR, is set to 1 on all entries to a given Exception level. This means:
  - Exceptions taken to the Exception level.
  - Exception returns to the Exception level.
  - It is IMPLEMENTATION DEFINED whether a reset into an Exception level generates an Exception Catch debug event.
- Might be taken synchronously, after the exception or reset entry or the exception return has been processed by the PE.
- Ignore the Execution state of the target Exception level.
- Might be ignored if halting is prohibited.

The EDECCR register contains fields that control when Exception Catch debug events are generated.

See Controlling Exception Catch debug events when FEAT_Debugv8p2 is implemented and Controlling Exception Catch debug events when FEAT_Debugv8p2 is not implemented.

For exception returns, the final Exception level of the exception return determines whether an Exception Catch debug event is generated. On an illegal exception return, an Exception Catch debug event is generated only if EDECCR is programmed to generate an Exception Catch debug event for an exception return to the current Exception level.

E3.4.1 Prioritization of Exception Catch debug events

The following rules define the prioritization of Exception Catch debug events:

- It is IMPLEMENTATION DEFINED whether Exception Catch debug events are higher or lower priority than Software Step exceptions and Halting Step debug events.
- Exception Catch debug events are higher priority than all synchronous exceptions other than Software Step exceptions and debug events other than Halting Step debug events.
- Exception Catch debug events are lower priority than Reset Catch debug events.
- When FEAT_Debugv8p2 is implemented, Exception Catch debug events are higher priority than pending asynchronous exceptions. Otherwise, the prioritization between asynchronous exceptions, asynchronous debug events, and an Exception Catch debug event, is IMPLEMENTATION DEFINED.

Note

As described in Prioritization of Synchronous exceptions taken to AArch64 state, an exception trapping form of a Vector Catch debug event might generate a second debug exception as part of the exception entry, before the Exception Catch debug event is taken. See Vector Catch exceptions or Vector Catch exceptions.

E3.4.2 Generating Exception Catch debug events when FEAT_Debugv8p2 is not implemented

When an Exception Catch debug event is generated after exception entry and halting is allowed at the target of the exception, the PE halts and enters Debug state:

- The PE halts and enters Debug state before the first instruction at the handler is executed, after the exception entry has updated the program counter, PSTATE, and syndrome registers for the exception.
- The PE does not fetch instructions from the vector address before entering Debug state, if address translation is disabled in the translation regime at the target Exception level.
- A second unmasked asynchronous exception can be taken before the PE enters Debug state. If this second exception does not generate an Exception Catch debug event, the exception handler executed at the higher Exception level later returns to the trapped Exception level, causing the Exception Catch debug event to be generated again.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When an Exception Catch debug event is generated on exception return and halting is allowed at the target of the exception return:

- There is no prioritization between asynchronous exceptions, asynchronous debug events, and an Exception Catch debug event generated on an exception return.
- The PE halts and enters Debug state after the exception return has updated the program counter and PSTATE, and before the execution of the first instruction at the return address is completed.

Otherwise, when the PE is executing code at a given Exception level, and the corresponding EDECCR bit is 1, it is CONSTRAINED UNPREDICTABLE whether an Exception Catch debug event is generated.

Examples of this are:

- If the debugger writes to EDECCR so that the current Exception level is trapped.
- If the OS restore code writes to OSECCR_EL1, OS Lock Exception Catch Control Register so that the current Exception level is trapped.

See also Debug state entry and debug event prioritization.

Note

It is possible to generate Exception Catch debug events:

- As a trap on all instruction fetches from the trapped Exception level as part of an instruction fetch.
- On entry to the Exception level, as described in Detailed Halting Step state machine behavior.

This is similar to the implementation options allowed for Vector Catch debug events. The architecture does not require that the event is generated following an `ISB` operation executed at the Exception level.

E3.4.3 Controlling Exception Catch debug events when FEAT_Debugv8p2 is implemented

When FEAT_Debugv8p2 is implemented the fields EDECCR.{NSR, SR, NSE, SE} control generation of Exception Catch debug events:

- On exception entry but not on exception return.
- On exception return but not on exception entry.
- On exception entry and exception return.

Exception entry, reset and exception return Exception Catch debug events are enabled as shown in Table E3-5.

Table E3-5 Summary of Exception Catch debug event control when FEAT_Debugv8p2 is implemented

(N)SR<n>	(N)SE<n>	Behavior on exception return to ELn	Behavior on exception taken to ELn, and if resets are Exception Catch debug events, reset into ELn
0	0	No action.	No action.
0	1	Halt if allowed.	Halt if allowed.
1	0	Halt if allowed.	No action.
1	1	No action.	Halt if allowed.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E3.4.4 Controlling Exception Catch debug events when FEAT_Debugv8p2 is not implemented

When FEAT_Debugv8p2 is not implemented, all Exception Catch debug events are enabled by a combination of the fields NSE and SE in EDECCR, as shown in Table E3-6.

Table E3-6 Summary of Exception Catch debug event control when FEAT_Debugv8p2 is not implemented

(N)SE<n>	Behavior on exception taken to ELn, return to ELn, and if resets are Exception Catch debug events, reset into ELn
0	No action.
1	Halt if allowed.

E3.4.5 Examples of Exception Catch debug events

If EDECCR == `0x0020`, meaning that the Exception Catch debug event is enabled for EL1, then the following exceptions generate Exception Catch debug events:

- An exception taken from EL0 to EL1.
- An exception return from EL2 to EL1.

For example, on taking a Data Abort exception from EL0 to EL1, using AArch64:

- ELR_EL1 and SPSR_EL1 are written with the preferred return address and PE state for a return to EL0.
- ESR_EL1 and FAR_EL1 are written with the syndrome information for the exception.
- DLR_EL0 is set to VBAR_EL1 + `0x400`, the synchronous exception vector.
- DSPSR_EL0 is written with the PE state for an exit to EL1.

The following do not generate Exception Catch debug events:

- An exception taken from EL0 to EL2.
- An exception return from EL2 to EL0.
- An exception taken from EL0 to EL1.

E3.4.6 Pseudocode description of Exception Catch debug events

The pseudocode functions `CheckExceptionCatch()` and `CheckPendingExceptionCatch()` are described in A-profile Architecture Pseudocode.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E3.5 External Debug Request debug event

External Debug Request debug events are asynchronous debug events.

An External Debug Request debug event can be generated by any of the following:

- When signaled by the embedded cross-trigger. See The Embedded Cross-Trigger Interface.
- The Performance Monitoring Unit. See PMU Overflow external debug request.
- An implementation might also support IMPLEMENTATION DEFINED ways of generating an External Debug Request debug event.

E3.5.1 Synchronization and External Debug Request debug events

For all Context synchronization events, if an External Debug Request debug event is asserted before the Context Synchronization event, and the External Debug Request debug event remains asserted and halting is allowed after the Context synchronization event, then the debug event is taken and the PE enters Debug state before the first instruction following the Context synchronization event completes its execution.

An External Debug Request debug event that is being asserted when the PE comes out of reset is taken, and the PE enters Debug state before the first instruction after the reset completes its execution, provided that halting is allowed when the PE exits reset state.

If the first instruction after the Context Synchronization event or after coming out of reset generates a synchronous exception then the architecture does not define the order in which the debug event and the exception or exceptions are taken.

Otherwise, when all of the following are true, External Debug Request debug events must be taken in finite time, without requiring the synchronization of any necessary change to the external authentication interface:

- Halting is allowed.
- The PE is not in a low-power state that is not required to exit on an external debug request. See Core power domain power states.

External debug request is a wakeup event for `WFI`, `WFE`, or `WFET` instructions. See Mechanisms for entering a low-power state.

Note

These rules are based on the rules that apply when taking asynchronous exceptions. See Asynchronous exception types.

If an unmasked External Debug Request debug event was pending but is changed to not pending before it is taken, then the architecture permits the External Debug Request debug event to be taken, but does not require this to happen. If the External Debug Request debug event is taken then it must be taken before the first Context Synchronization event after the External Debug Request debug event was changed to not pending.

Example E3-3 shows an example of the synchronization requirements.

Example E3-3 Synchronization requirements

Secure software locks up in a tight loop, so it executes indefinitely without any synchronization operations. An External debug request must be able to break the PE out of that loop. This is a requirement even if **DBGEN** or **SPIDEN** or both are LOW on entry to the loop, meaning that halting is prohibited, and are only asserted HIGH later.

E3.5.2 PMU Overflow external debug request

When the PE halts and enters Debug state because of a PMU Overflow external debug request:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- DLR_EL0 is set to the address of the first instruction that must be executed on exit from Debug state. DSPSR_EL0 is correct for this instruction.
- EDSCR.STATUS is set to `0b010011`, external debug request.

E3.5.3 Pseudocode description of External Debug Request debug events

The `ExternalDebugRequest()` function is described in R-profile Architecture Pseudocode.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E3.6 OS Unlock Catch debug event

An OS Unlock Catch debug event is generated when enabled and the state of the OS Lock changes from locked to unlocked. When FEAT_DoPD is implemented, CTIDEVCTL.OSUCE enables an OS Unlock Catch debug event, otherwise EDECR.OSUCE enables an OS Unlock Catch debug event.

When the OS Lock is unlocked, the PE sets EDESR.OSUC to 1 if the OS Unlock Catch debug event is enabled, and the PE is in Non-debug state, meaning the OS Unlock Catch debug event becomes pending. However, this is an indirect write to EDESR.OSUC, meaning the OS Unlock Catch debug event is not guaranteed to be taken before a subsequent Context synchronization event. If the PE enters Debug state or the OS Unlock Catch debug event is disabled before EDESR.OSUC becomes set to 1, then EDESR.OSUC might not be set.

OS Unlock Catch debug events are not generated if the OS Lock is unlocked when the PE is in Debug state. See also Synchronization and Halting debug events.

EDESR.OSUC is cleared to 0 on a Warm reset and on exiting Debug state.

E3.6.1 Using the OS Unlock Catch debug event

When the Core power domain is completely off or in a low-power state, a debugger is permitted to access a debug register that is implemented in the External debug power domain. However, if a debugger attempts to access a debug register that is implemented in the Core power domain when the Core power domain registers cannot be accessed, and that access returns an error, the debugger must retry the access.

Regularly powering down the Core power domain can result in unreliable debugger behavior.

The debugger can program a Reset Catch debug event to halt the PE when it has powered up, and can program the debug registers from Debug state. However, if the PE boot software restores the debug registers, as described in Debug OS Save and Restore sequences, then newly written values are overwritten by the restore sequence.

The debugger can program an OS Unlock Catch debug event to halt the PE after the restore sequence has completed, and program the debug registers from Debug state.

E3.6.2 Pseudocode description of OS Unlock Catch debug event

The `CheckOSUnlockCatch()` function is called when the OS Lock is unlocked.

The `CheckPendingOSUnlockCatch()` function is called before an instruction is executed. If an OS Unlock Catch is pending, it generates the debug event.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E3.7 Reset Catch debug events

A Reset Catch debug event is generated when enabled, and the PE exits reset state. When the Reset Catch debug event is generated, it is recorded by setting EDESR.RC to 1. When FEAT_DoPD is implemented, CTIDEVCTL.RCE enables a Reset Catch debug event, otherwise EDECR.RCE enables a Reset Catch debug event.

If halting is allowed when the event is generated, the Reset Catch debug event is taken immediately and synchronously. On entering Debug state, DLR_EL0, Debug Link Register has the address of the reset vector. The PE must not fetch any instructions from memory.

Otherwise, the Reset Catch debug event is pended and taken when halting is allowed. See Synchronization and Halting debug events for more information.

This means that EDESR.RC is set to the value of EDECR.RCE or CTIDEVCTL.RCE on a Warm reset. EDESR.RC is cleared to 0 on exiting Debug state.

E3.7.1 Pseudocode description of Reset Catch debug event

The `CheckResetCatch()` function is called after reset before executing any instruction.

The `CheckPendingResetCatch()` function is called before an instruction is executed. If a Reset Catch is pending, it generates the Reset Catch debug event.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E3.8 Software Access debug event

When the value of EDSCR.TDA is 1, software access to the following AArch64 debug System registers generate a Software Access debug event:

- The Breakpoint Value Registers, DBGBVR<n>_EL1, Debug Breakpoint Value Registers, n = 0 - 63.
- The Breakpoint Control Registers, DBGBCR<n>_EL1, Debug Breakpoint Control Registers, n = 0 - 63.
- The Watchpoint Value Registers, DBGWVR<n>_EL1, Debug Watchpoint Value Registers, n = 0 - 63.
- The Watchpoint Control Registers, DBGWCR<n>_EL1, Debug Watchpoint Control Registers, n = 0 - 63.

However, EDSCR.TDA is ignored if any of the following applies:

- The value of OSLSR_EL1, OS Lock Status Register.OSLK == 1, meaning that the OS Lock is locked.
- Halting is prohibited. See Halting allowed and halting prohibited.
- The register access generates a higher priority synchronous exception or debug event.

Note

- The only accesses to the specified registers that generate a Software Access debug event are accesses to System registers in AArch64 state.
- Accesses by a PE using the external debug interface never generate a Software Access debug event.

E3.8.1 Pseudocode description of Software Access debug event

The `CheckSoftwareAccessToDebugRegisters()` function is described in R-profile Architecture Pseudocode.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E3.9 Synchronization and Halting debug events

The behavior of external debug depends on:

- Indirect reads of:
  - External debug registers.
  - System registers, including system debug registers.
  - Special-purpose registers.
- The state of the external authentication interface.

For some registers, all read and write accesses that update the register occur in program order, without any additional synchronization, but others require an explicit Context Synchronization event. For more information on the synchronization of register updates, see:

- Synchronization requirements for AArch64 System registers.
- Synchronization of changes to the external debug registers.

Changes to the external authentication interface do not require explicit synchronization to affect External Debug Request debug events. See Synchronization and External Debug Request debug events.

For changes that require explicit synchronization, it is CONSTRAINED UNPREDICTABLE whether instructions between the change and the Context Synchronization event observe the old state or the new state.

This means that any change to these registers or the external authentication interface requires explicit synchronization by a Context Synchronization event before the change takes effect. This ensures that for instructions appearing in program order after the change, the change affects the following:

- The generation and behavior of Breakpoint and Watchpoint debug events. See Synchronization and debug exceptions for exceptions taken from AArch64 state.
- The generation of all Halting debug events by instructions.
- Taking a pending Halting debug event or other asynchronous debug event. See:
  - Pending Halting debug events.
  - Synchronization and External Debug Request debug events.
- The behavior of the Halting Step state machine. See Synchronization and the Halting Step state machine.

E3.9.1 Pending Halting debug events

A Halting debug event might be pending:

- If Halting Step of an instruction sets EDESR.SS to 1, and halting is prohibited following the step, then the Halting Step state machine is inactive but a Halting Step debug event is pending.
- If a Reset Catch debug event sets EDESR.RC to 1, and halting is prohibited following reset, then a Reset Catch debug event is pending.
- If an OS Unlock Catch debug event sets EDESR.OSUC to 1, then an OS Unlock Catch debug event is pending.

Pending Halting debug events are taken asynchronously when halting is allowed.

Pending Halting debug events are discarded by a Cold reset. The debugger can also force a pending event to be dropped by writing to EDESR.

Any Halting debug event that is observed as pending in the EDESR before a Context Synchronization event is taken and the PE enters Debug state before the first instruction following the Context Synchronization event completes its execution. This is possible only if halting is allowed after completion of the Context Synchronization event.

If the first instruction after the Context Synchronization event generates a synchronous exception then the architecture does not define the order in which the debug event and the exception or exceptions are taken, unless a Halting Step debug event is pending and EDESR.SS == 1.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter E4
The Debug Communication Channel and Instruction Transfer Register

This chapter describes communication between a debugger and the implemented debug logic, using the *Debug Communications Channel* (DCC) and the *Instruction Transfer Register* (ITR), and associated control flags. It contains the following sections:

- Introduction.
- DCC and ITR registers.
- DCC and ITR access modes.
- Flow control of the DCC and ITR registers.
- Synchronization of DCC and ITR accesses.
- Interrupt-driven use of the DCC.
- Pseudocode description of the operation of the DCC and ITR registers.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E4.1 Introduction

The *Debug Communications Channel*, DCC, is a channel for passing data between the PE and an external agent, such as a debugger. The DCC provides a communications channel between:

- An external debugger, described as the *debug host*.
- The debug implementation on the PE, described as the *debug target*.

The DCC can be used:

- As a 32-bit full-duplex channel.
- As a 64-bit half-duplex channel.

The DCC is an essential part of Debug state operation and can also be used in Non-debug state.

The *Instruction Transfer Register*, ITR, passes instructions to the PE to execute in Debug state.

The PE includes flow-control mechanisms for both the DCC and ITR.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E4.2 DCC and ITR registers

The DCC comprises *data transfer registers*, the DTRs, and associated flow-control flags. The data transfer registers are DTRRX and DTRTX.

The ITR comprises a single register, EDITR, and associated flow-control flags.

In AArch64 state, software can access the data transfer registers as:

- A receive and transmit pair for 32-bit full duplex operation:
  - The write-only DBGDTRTX_EL0 register to transmit data.
  - The read-only DBGDTRRX_EL0 register to receive data.
- A single 64-bit read/write register, DBGDTR_EL0, for 64-bit half-duplex operation.
- The read/write OSDTRTX_EL1 and OSDTRRX_EL1 registers for save and restore.

The data transfer registers are also accessible by the external debug interface as a pair of 32-bit registers, DBGDTRRX_EL0 and DBGDTRTX_EL0. Both registers are read/write, allowing both 32-bit full-duplex and 64-bit half-duplex operation.

The DCC flow-control flags are EDSCR.{RXfull, TXfull, RXO, TXU}:

- The RXfull and TXfull ready flags are used for flow-control and are visible to software in the Debug system registers in MDCCSR_EL0, Monitor DCC Status Register.
- The RX overrun flag, RXO, and the TX underrun flag, TXU, report flow-control errors.
- The flow-control flags are also accessible by software as simple read/write bits for saving and restoring over a powerdown when the OS Lock is locked in MDSCR_EL1, Monitor Debug System Control Register.
- The flow-control flags are accessible from the external debug interface in EDSCR.

Figure E4-1 shows the System register and external debug interface views of the EDSCR and DTR registers. These figures do not include the save and restore views.

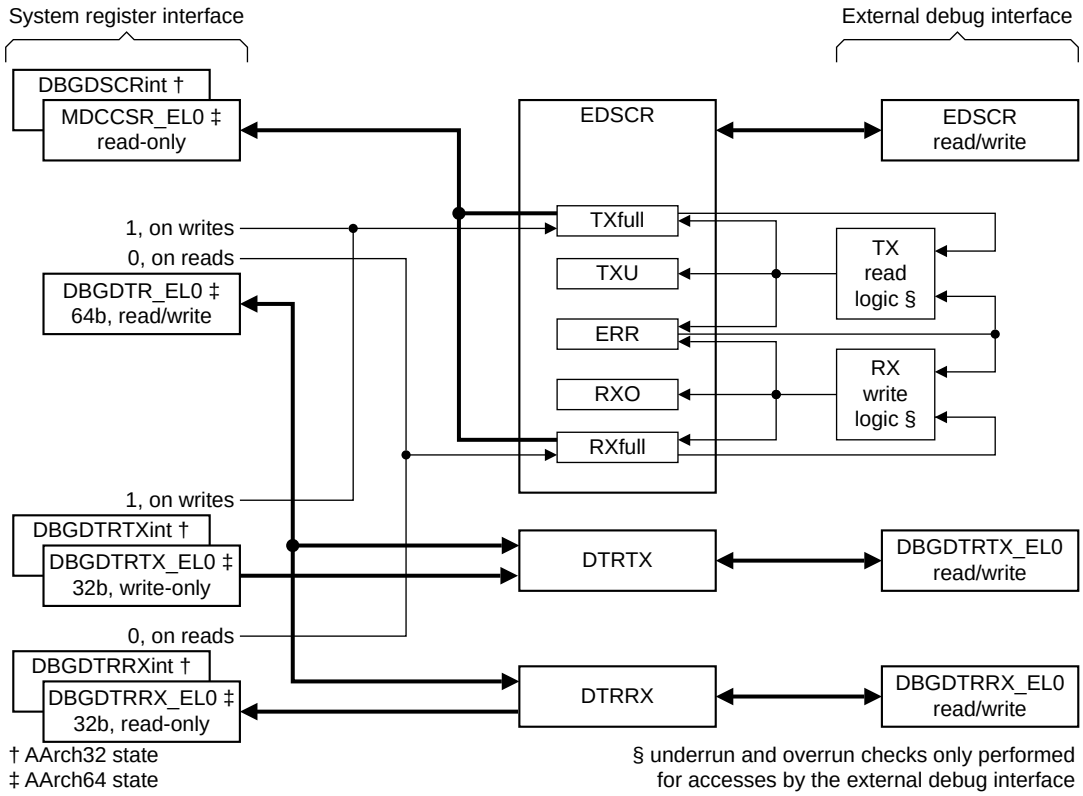


Figure E4-1 System register and external debug interface views of EDSCR and DTR registers, Normal access mode

EDITR and the ITR flow-control flags, EDSCR.{ITE, ITO} are accessible only by the external debug interface:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The EDITR specifies an instruction to execute in Debug state.
- The ITR empty flag, ITE, is used for flow-control.
- The ITR overrun flag, ITO, reports flow-control errors.

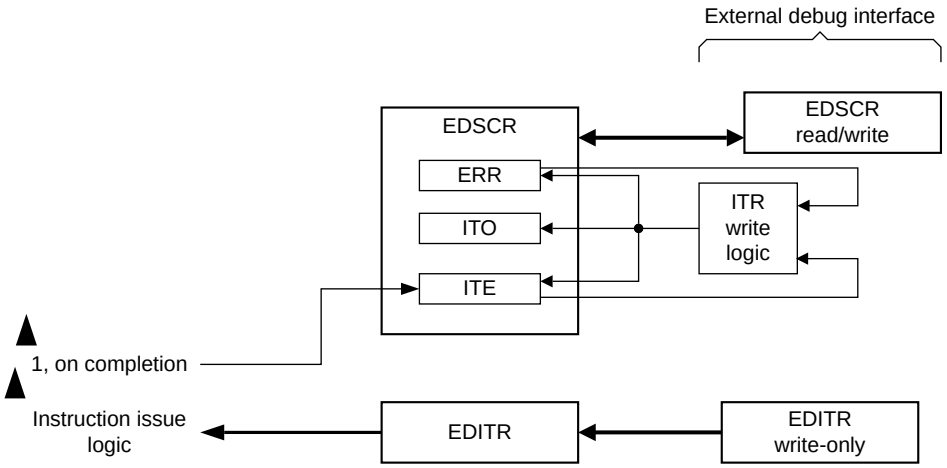


Figure E4-2 External debug interface views of EDSCR and EDITR registers, Normal access mode

The sticky overflow flag, EDSCR.ERR, is used by both the DCC and ITR to report flow-control errors.

To save and restore the DCC registers for an external debugger over powerdown, software uses:

- The MDSCR_EL1, OSDTRTX_EL1, and OSDTRRX_EL1 registers in AArch64 state.

Note

There is no save and restore mechanism for the ITR registers as the ITR is used only in Debug state.

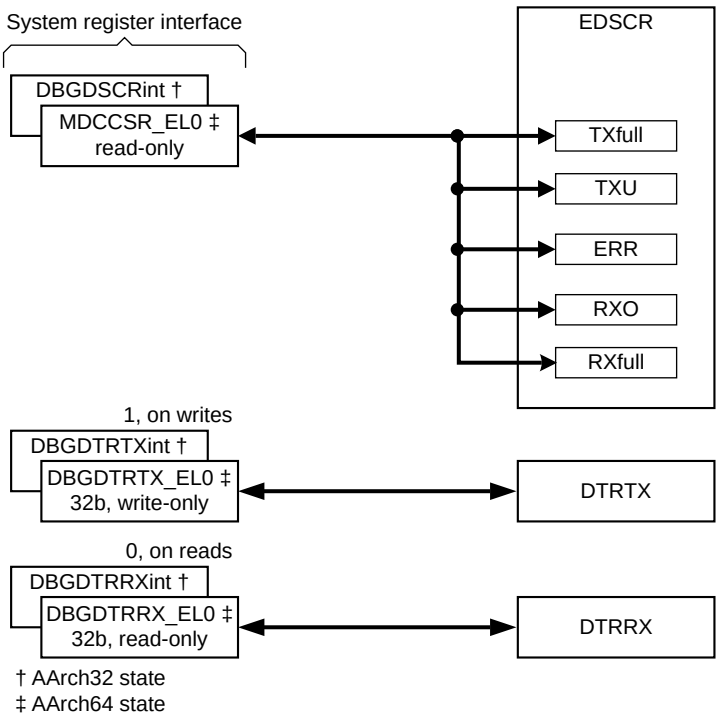


Figure E4-3 System register views of EDSCR and DTR registers for save and restore

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E4.3 DCC and ITR access modes

The DCC and ITR support two access modes:

DCC and ITR access mode, links to description	Applies when:
Normal access mode	EDSCR.MA == 0 or the PE is in Non-debug state
Memory access mode	EDSCR.MA == 1 and the PE is in Debug state

E4.3.1 Normal access mode

The Normal access mode allows use of the DCC as a communications channel between target and host. It also allows the use of the ITR for issuing instructions to the PE in Debug state.

In Normal access mode, if there is no overrun or underrun, the following occurs:

For accesses by software:

- Direct writes to DBGDTRTX_EL0, Debug Data Transfer Register, Transmit update the value in DTRTX and indirectly write 1 to TXfull.
- Direct reads from DBGDTRRX_EL0, Debug Data Transfer Register, Receive return the value in DTRRX and indirectly write 0 to RXfull.
- Direct writes to DBGDTR_EL0 update both DTRTX and DTRRX, indirectly write 1 to TXfull, and do not change RXfull:
  - DTRTX is set from bits[31:0] of the transfer register.
  - DTRRX is set from bits[63:32] of the transfer register.
- In AArch64 state, direct reads from DBGDTR_EL0 return the concatenation of DTRRX and DTRTX, indirectly write 0 to RXfull, and do not change TXfull:
  - Bits[31:0] of the transfer register are set from DTRRX.
  - Bits[63:32] of the transfer register are set from DTRTX.

Note

For DBGDTR_EL0, the word order is reversed for reads with respect to writes.

Software reads TXfull and RXfull using MDCCSR_EL0, Monitor DCC Status Register.

For accesses by the external debug interface:

- Writes to EDITR trigger the instruction to be executed if the PE is in Debug state, if the PE is in AArch64 state, this is an A64 instruction.
- Reads of DBGDTRTX_EL0 return the value in DTRTX and indirectly write 0 to TXfull.
- Writes to DBGDTRTX_EL0 update the value in DTRTX and do not change TXfull.
- Reads of DBGDTRRX_EL0 return the value in DTRRX and do not change RXfull.
- Writes to DBGDTRRX_EL0 update the value in DTRRX and indirectly write 1 to RXfull.
  TXfull and RXfull are visible to the external debug interface in EDSCR.
  The PE detects overrun and underrun by the external debug interface, and records errors in EDSCR.{TXU, RXO, ITO, ERR}. See Flow control of the DCC and ITR registers.

See also Synchronization of DCC and ITR accesses.

E4.3.2 Memory access mode

When the PE is in Debug state, *Memory access mode* can be selected to accelerate word-aligned block reads or writes of memory by an external debugger. Memory access mode can be enabled only in Debug state, and no instructions can be issued directly by the debugger when in Memory access mode.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If there is no overrun or underrun when in Memory access mode, an access by the external debug interface results in the following:

- External reads from DBGDTRTX_EL0 cause:
  1. The existing value in DTRTX to be returned. This clears EDSCR.TXfull to 0.
  2. The equivalent of `LDR W1,[X0],#4` to be executed.
  3. The equivalent of the `MSR DBGDTRTX_EL0,X1` instruction to be executed.
  4. EDSCR.{TXfull, ITE} to be set to {1,1}, and X1 or R1 to be set to an UNKNOWN value.
- External writes to DBGDTRRX_EL0 cause:
  1. The value in DTRRX to be updated. This sets EDSCR.RXfull to 1.
  2. The equivalent of the instruction `MRS X1,DBGDTRRX_EL0` to be executed.
  3. The equivalent of the instruction `STR W1,[X0],#4` to be executed.
  4. EDSCR.{RXfull, ITE} to be set to {0,1}, and X1 or R1 to be set to an UNKNOWN value.
- External reads from DBGDTRRX_EL0 return the last value written to DTRRX.
- External writes to EDITR generate an overrun error.

During these accesses, EDSCR.{TXfull, RXfull, ITE} are used for flow control.

Note

An overrun or underrun might result in EDSCR.ERR being set to 1 asynchronously to the sequence of operations that are outlined in this section. As this is timing-dependent, it is UNPREDICTABLE when the EDSCR.ERR flag affects the instructions and therefore whether neither instruction, only the first instruction, or both instructions are executed. If the second instruction is executed, then the first instruction must have been executed. However, in each case X1 or R1 is set to an UNKNOWN value. This means that:

- In both cases, if the memory access instruction is not executed, then the base register X0 or R0 is not updated, meaning the debugger can determine the last accessed location.
- In the list describing External reads from DBGDTRTX_EL0, DTRTX and EDSCR.TXfull get set to UNKNOWN values. If the load was executed, then the value that was read by the PE is lost. This means the operation might need to be repeated by the debugger, and it is not advisable to use Memory access mode to read from read-sensitive locations using the underrun and overrun detection for flow control.
- In the list describing External writes to DBGDTRRX_EL0, EDSCR.RXfull is set to an UNKNOWN value.

A Data Abort from the memory access can also set EDSCR.ERR to 1. See Data Aborts in Memory access mode.

The architecture does not require precisely when these flags are set or cleared by the sequence of operations outlined in this section. For example, in the case of an external write to DBGDTRRX_EL0, in AArch64 state, RXfull might be cleared after step 2, or it might not be cleared until after step 3, as an implementation is free to fuse these steps into a single operation. The architecture does require that the flags are set as at step 4 when the PE is ready to accept a further read or write without causing an overrun error or an underrun error.

The process outlined in this section represents a simple sequential execution model of Memory access mode. An implementation is free to pipeline, buffer, and reorder instructions and transactions, as long as the following remain true:

- Data items are transferred into and out of the DTR in order and without loss of data, other than as a result of an overrun or an underrun.
- Data Aborts occur in order.
- The constraints of the memory type are met.
- In the list describing External reads from :
  - The `MSR` equivalent operation at step 3 of the sequence reads the value loaded by step 2.
  - If the list is performed in a loop, for all but the first iteration of this list, the value read by step 1 returns the values written by the `MSR` equivalent operation at the previous iteration of step 3.
- In the list describing External writes to :
  - The `MRS` equivalent operation at step 2 of the sequence returns the value written at step 1.
  - The `STR` equivalent at step 3 of the sequence writes the value read at step 2.
- If the PE cannot accept a read or write, as applicable, during the sequence, then the flags are updated to indicate an overrun or underrun.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

See Flow control of the DCC and ITR registers for more information on overrun and underrun.

E4.3.2.1 Ordering, access sizes and effect on Exclusives monitors

For the purposes of memory ordering, access sizes, and effect on the Exclusives monitor, accesses in Memory access mode are consistent with load/store word instructions executed by the PE.

The simple sequential access model of Memory access mode, as stated in Memory access mode, must also be ordered with respect to instructions executed as a result of External writes to EDITR in Normal mode both before and after accesses to the DTR registers in Memory access mode.

E4.3.2.2 Data Aborts in Memory access mode

If a memory access generates a Data Abort, then:

- The Data Abort exception is taken. See Exceptions in Debug state:
  - This means EDSCR.ERR is set to 1, see Cumulative error flag.
  - If the Data Abort occurs on stage 2 of an address translation, then the values returned in the ISV field and in bits[23:14] of the ISS are UNKNOWN.
    If this Data Abort is taken to EL2, the ISS is returned by ESR_EL2. ISS encoding for an exception from a Data Abort describes the usual encoding of this ISS.
- Register R0 retains the address that generated the abort.
- Register R1 is set to an UNKNOWN value.
- EDSCR.TXfull, for a load, or EDSCR.RXfull, for a store, is set to an UNKNOWN value.
- DTRTX, for a load, or DTRRX, for a store, is set to an UNKNOWN value.
- EDSCR.ITE is set to 1.

E4.3.2.3 Illegal Execution state exception

If PSTATE.IL is set to 1 when EDSCR.MA == 1, then on an external write access to DBGDTRRX_EL0 or an external read from DBGDTRTX_EL0, it is CONSTRAINED UNPREDICTABLE whether the PE:

- Takes an Illegal Execution state exception without performing any operations. In this case:
  - EDSCR.ERR is set to 1, see Cumulative error flag.
  - Register R0 is unchanged.
  - Register R1 is set to an UNKNOWN value.
  - EDSCR.TXfull or EDSCR.RXfull, as applicable, is set to an UNKNOWN value.
  - DTRTX or DTRRX, as applicable, is set an UNKNOWN value.
  - EDSCR.ITE is set to 1.

  See also Exceptions in Debug state.
- Ignores PSTATE.IL.

Note

The typical usage model for Memory access mode involves executing instructions in Normal access mode to set up X0 before setting EDSCR.MA to 1. These instructions generate an Illegal state exception if PSTATE.IL is set to 1.

E4.3.2.4 Alignment constraints

If the address in R0 is not aligned to a multiple of four, the behavior is as follows:

- For each external DTR access a CONSTRAINED UNPREDICTABLE choice of:
  1. The PE makes an unaligned memory access to R0. If alignment checking is enabled for the memory access, this generates an Alignment fault.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2. The PE makes a memory access to `Align(X[0],4)`.
3. The PE generates an Alignment fault, regardless of whether alignment checking is enabled.
4. The PE does nothing.

- Following each memory access, if there is no Data Abort, R0 is updated with an UNKNOWN value.
- For external writes to DBGDTRRX_EL0, if the PE writes to memory, an UNKNOWN value is written.
- For external reads of DBGDTRTX_EL0 an UNKNOWN value is returned.
- The RXfull and TXfull flags are left in an UNKNOWN state, meaning that a DBGDTRTX_EL0 read can trigger a TX underrun, and a DBGDTRTX_EL0 write can trigger an RX overrun.

E4.3.3 Memory-mapped accesses to the DCC and ITR

Writes to the flags in EDSCR by external debug interface accesses to the DCC and the ITR registers are indirect writes, because they are a side-effect of the access. The indirect write might not occur for a memory-mapped access to the external debug interface. For more information, see Register access permissions for memory-mapped accesses.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E4.4 Flow control of the DCC and ITR registers

- Ready flags.
- Buffering writes to EDITR.
- Overrun and underrun flags.
- Cumulative error flag.

E4.4.1 Ready flags

In Normal access mode:

- For the DTR registers there are two ready flags:
  - EDSCR.RXfull == 1 indicates that DBGDTRRX_EL0 contains a valid value that has been written by the external debugger and not yet read by software running on the target.
  - EDSCR.TXfull == 1 indicates that DBGDTRTX_EL0 contains a valid value that has been written by software running on the target and not yet read by an external debugger.
- For the ITR register there is a single ready flag:
  - EDSCR.ITE == 1 indicates that the PE is ready to accept an instruction to the ITR.

Note

The architecture permits a PE to continue to accept and buffer instructions when previous instructions have not completed their architecturally defined behavior, as long as those instructions are discarded if EDSCR.ERR is 1, either by an underrun or overrun or by any of the other error conditions described in this architecture, such as an instruction generating an abort.

In Memory access mode:

- EDSCR.{RXfull, ITE} == {0,1} indicates that DBGDTRRX_EL0 is empty and the PE is ready to accept a word external write to DBGDTRRX_EL0.
- EDSCR.{TXfull, ITE} == {1,1} indicates that DBGDTRTX_EL0 is full and the PE is ready to accept a word external read from DBGDTRTX_EL0.

All other values indicate that the PE is not ready, and result in a DTR overrun or underrun error, an ITR overrun error, or both, as defined in Overrun and underrun flags.

EDSCR.{ITE, RXfull, TXfull} shows the status of the ITR and DCC registers. It ignores the question of whether a read or write cannot be accepted because, for example, EDSCR.ERR is set or the OPTIONAL Software Lock is locked for memory-mapped accesses (EDLSR.SLK == 1).

E4.4.2 Buffering writes to EDITR

The architecture permits a processor to continue to accept and buffer instructions when previous instructions have not completed their architecturally defined behavior, provided that:

- Those instructions are discarded if EDSCR.ERR is set to 1, either by an underrun or an overrun, or by any other error conditions described in this architecture, such as an instruction generating an abort.
- The PE maintains the simple sequential execution model with the order of instructions determined by the order in which the PE accepts the EDITR writes. In particular, the buffered instructions must be executed in the Execution state consistent with a simple sequential execution of the instructions, even if one of the previous instructions is a state changing operation, such as `DCPS` or `DRPS`.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E4.4.3 Overrun and underrun flags

Each of the ready flags has a corresponding overrun or a corresponding underrun flag. These are sticky status flags that are set if the register is accessed using the external debug interface when the corresponding ready flag is not in the ready state.

If the PE is in Debug state and Memory access mode, the corresponding error flag is also set if the PE is not ready to accept an operation because a previous load or store is still in progress. The sticky status flag remains set until cleared by writing 1 to EDRCR.CSE.

Note

The architecture permits a PE to continue to accept and buffer data to write to memory in Memory access mode.

Table E4-2 shows DCC and ITR ready flags and the overrun and underrun flags associated with them.

Table E4-2 DCC and ITR ready flags and the associated overrun/underrun flags

External debug interface access	Overrun/Underrun condition	EDSCR flag
Write DBGDTRRX_EL0	`EDSCR.RXfull == '1' || (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0')`	RXO
Read DBGDTRTX_EL0	`EDSCR.TXfull == '0' || (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0')`	TXU
Write EDITR	`Halted() && (EDSCR.ITE == '0' || EDSCR.MA == '1')`	ITO

When an overrun or underrun flag is set to 1, the cumulative error flag, EDSCR.ERR, described in Cumulative error flag, is also set to 1.

In the event of an external write to DBGDTRRX_EL0 or EDITR generating an overrun, or an external read from DBGDTRTX_EL0 generating an underrun:

- For a write, the written value is ignored.
- For a read, an UNKNOWN value is returned.
- EDSCR.TXfull, EDSCR.RXfull or EDSCR.ITE, as applicable, are not updated.

There is no overrun or underrun detection on external reads of DBGDTRRX_EL0 or external writes of DBGDTRTX_EL0.

There is no overrun or underrun detection of direct reads and direct writes of the DTR System registers by software:

- If RXfull == 0, a direct read of DBGDTRRX_EL0, Debug Data Transfer Register, Receive or DBGDTR_EL0 returns UNKNOWN.
- If TXfull == 1, a direct write of:
  - DBGDTRTX_EL0, Debug Data Transfer Register, Transmit sets DTRTX to UNKNOWN.
  - DBGDTR_EL0 sets DTRRX and DTRTX to UNKNOWN.

See DCC accesses in Non-debug state for more information.

E4.4.3.1 Accessing 64-bit data

In AArch64 state, a software access to the DBGDTR_EL0 register and an external debugger access to both DBGDTRRX_EL0 and DBGDTRTX_EL0 can perform a 64-bit half-duplex operation.

However, there is only overrun and underrun detection on one of the external debug registers. That is:

- If software directly writes a 64-bit value to DBGDTR_EL0, only TXfull is set to 1, meaning:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - A subsequent external write to DBGDTRRX_EL0 would not be detected as an overrun.
  - If the external debugger reads DBGDTRTX_EL0 first, software might observe MDCCSR_EL0.TXfull == 0 and send a second value before the external debugger reads DBGDTRRX_EL0, leading to an undetected overrun.
- On external writes to both DBGDTRRX_EL0 and DBGDTRTX_EL0, only RXfull is set to 1, meaning:
  - A subsequent direct write of DBGDTRTX_EL0 would not be detected as an overrun.
  - If the external debugger writes to DBGDTRRX_EL0 first, software might observe MDCCSR_EL0.RXfull == 1 and read a full 64-bit value, before the external debugger writes to DBGDTRTX_EL0, leading to an undetected underrun.

To avoid this, debuggers need to be aware of the data size used by software for transfers and ensure that 64-bit data is read or written in the correct order. If the PE is in Non-debug state, this order is as follows:

- The external debugger must check EDSCR.{RXfull, TXfull} before each transfer.
- To receive a 64-bit value from the target, the external debugger must read DBGDTRRX_EL0 before reading DBGDTRTX_EL0.
- To send a 64-bit value to the target, the external debugger must write to DBGDTRTX_EL0 before writing DBGDTRRX_EL0.

Because three accesses are required to transfer 64 bits of data, 64-bit transfers are not recommended for regular communication between host and target. The use of underrun and overrun detection means that only one access is required for 32 bits of data when using 32-bit transfers.

In Debug state, the debugger controls the instructions executed by the PE, so these limitations do not apply. 64-bit transfers provide a means to transfer a 64-bit general register between the host and the target in Debug state.

E4.4.4 Cumulative error flag

The cumulative error flag, EDSCR.ERR, is set to 1:

- On taking an exception from Debug state.
- On any signaled overrun or underrun in the DCC or ITR.

When EDSCR.ERR == 1:

- External reads of DBGDTRTX_EL0 do not have any side-effects.
- External writes to DBGDTRRX_EL0 are ignored.
- External writes to EDITR are ignored.
- No further instructions can be issued in Debug state. This includes any instructions previously accepted as external writes to EDITR that occur in program order after the instruction or access that caused the error.

This allows a debugger to stream data, or, in Debug state, instructions, to the target without having to:

- Check EDSCR.{RXfull, TXfull, ITE} before each access.
- Check EDSCR.{ITO, RXO, TXU} following each access, for overrun or underrun.
- Check PSTATE or other syndrome registers, or both, for an exception following each instruction executed in Debug state that might generate a synchronous exception.

The cumulative error flag remains set until cleared to 0 by writing 1 to EDRCR.CSE. However, the effect of writing 1 to EDRCR.CSE to clear EDSCR.ERR is CONSTRAINED UNPREDICTABLE when both of the following apply:

- The PE is in Debug state.
- The value of EDSCR.ITE is 0.

When these conditions apply and a value of 1 is written to EDRCR.CSE, either or both of the following might occur:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- EDSCR.ERR is not cleared to 0.
- Any instructions in EDITR that have not been executed might be executed subsequently, rather than being ignored.

Note

This means that a debugger must poll EDSCR.ITE until it has the value 1, indicating that EDITR is empty, before writing to EDRCR.CSE to clear the EDSCR.ERR flag to 0.

For overruns and underruns, EDSCR.{ITO, RXO, TXU} record the error type.

E4.4.4.1 Pseudocode description of clearing the error flag

The `ClearStickyErrors()` pseudocode function is described in R-profile Architecture Pseudocode.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E4.5 Synchronization of DCC and ITR accesses

In addition to the standard synchronization requirements for register accesses, the following subsections describe additional requirements that apply for the DCC and ITR registers:

- Summary of System register accesses to the DCC.
- DCC accesses in Non-debug state.
- Synchronization of DCC interrupt request signals.
- DCC and ITR access in Debug state.

In these sections, accesses by the external debug interface are referred to as external reads and external writes. Accesses to System registers are referred to as direct reads, direct writes, indirect reads, and indirect writes.

Note

In Synchronization requirements for AArch64 System registers external reads and external writes are described as forms of indirect access. This whole section uses more explicit terminology.

The DTR registers and the DCC flags, TXfull and RXfull, form a communication channel, with one end operating asynchronously to the other. Implementations must respect the ordering of accesses to these registers in order to maintain the correct behavior of the channel.

External reads of, and external writes to DBGDTRRX_EL0 and DBGDTRTX_EL0 are asynchronous to direct reads of, and direct writes to, DBGDTRRX_EL0, Debug Data Transfer Register, Receive, DBGDTRTX_EL0, Debug Data Transfer Register, Transmit, and in AArch64 state DBGDTR_EL0, made by software using System register access instructions. The direct reads and direct writes indirectly write to the DCC flags. The external reads and external writes indirectly read the DCC flags to check for underrun and overrun.

Throughout this section:

DCC flags Means any or all of the following:

- The EDSCR.{RXfull.TXfull} ready flags.
- The EDSCR.RXO overrun flag.
- The EDSCR.TXU underrun flag.
- The EDSCR.ERR cumulative error flag.

ITR flags Means any or all of the following:

- The EDSCR.ITE ready flag.
- The EDSCR.ITO overrun flag.
- The EDSCR.ERR cumulative error flag.

E4.5.1 Summary of System register accesses to the DCC

System register accesses to the DTR registers are direct reads and writes of those registers, as shown in Table E4-3. Several of these instructions access the same registers using different encodings.

DBGDTRRX_EL0 and DBGDTRTX_EL0 are encoded as MRS and MSR accesses respectively to the same System register, even though they access different underlying register values. DBGDTRRX_EL0, Debug Data Transfer Register, Receive and DBGDTRTX_EL0, Debug Data Transfer Register, Transmit are similarly encoded as MRC and MCR accesses respectively to the same System register. The encoding means that direct reads and writes using these encodings must be ordered with respect to each other. For more information, see Synchronization requirements for AArch64 System registers.

Table E4-3 shows a summary of System register accesses to the DCC.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table E4-3 Summary of System register accesses to the DCC

Operation	OS Lock	AArch64 (MRS/MSR)	Description
Read	-	DBGDTRRX_EL0	Direct read of DTRRX. Indirect write to the DCC flags. An STC instruction that reads DBGDTRRXint makes an indirect write to DBGDSCRint.RXfull.
Write	-	DBGDTRTX_EL0	Direct read of DTRTX. Indirect write to the DCC flags. An LDC instruction that writes to DBGDTRTXint using a value read from memory is a direct write to DBGDTRTXint.
Read/write	-	DBGDTR_EL0	Direct read/write of both DTRRX and DTRTX. Indirect write to the DCC flags.
Read	-	MDCCSR_EL0	Direct read of the DCC flags.
Read/write	-	OSDTRRX_EL1	Direct read/write of DTRRX.
Read/write	-	OSDTRTX_EL1	Direct read/write of DTRTX.
Read	Unlocked	MDSCR_EL1	Direct read of DCC flags.
Read/write	Locked	MDSCR_EL1	Direct read/write of DCC flags.

E4.5.2 DCC accesses in Non-debug state

In Non-debug state DCC accesses are as described in Normal access mode:

- If a direct read of MDCCSR_EL0, Monitor DCC Status Register returns RXfull == 1, then a following direct read of DBGDTRRX_EL0, Debug Data Transfer Register, Receive, or in AArch64 state of DBGDTR_EL0, returns valid data and indirectly writes 0 to MDCCSR_EL0, Monitor DCC Status Register.RXfull as a side-effect.
- If a direct read of MDCCSR_EL0, Monitor DCC Status Register returns TXfull == 0, then a following direct write to DBGDTRTX_EL0, Debug Data Transfer Register, Transmit, or in AArch64 state to DBGDTR_EL0, writes the intended value, and indirectly writes 1 to MDCCSR_EL0, Monitor DCC Status Register.TXfull as a side-effect.

No Context Synchronization event is required between these two instructions. Overrun and underrun detection prevents intervening external reads and external writes affecting the outcome of the second instruction.

The indirect write to the DCC flags as part of the DTR access instruction is made atomically with the DTR access.

Because a direct read of DBGDTRRX_EL0, Debug Data Transfer Register, Receive is an indirect write to MDCCSR_EL0, Monitor DCC Status Register.RXfull, it must occur in program order with respect to the direct read of MDCCSR_EL0, Monitor DCC Status Register, meaning it must not return a speculative value for DTTRX that predates the RXfull flag returned by the read of MDCCSR_EL0, Monitor DCC Status Register. The direct write to DBGDTRTX_EL0, Debug Data Transfer Register, Transmit must not be executed speculatively.

Direct reads of DBGDTRRX_EL0, Debug Data Transfer Register, Receive, or in AArch64 state DBGDTR_EL0, and MDCCSR_EL0, Monitor DCC Status Register, must occur in program order with respect to other direct reads of the same register using the same encoding.

The following accesses have an implied order within the atomic access:

- In the simple sequential execution of the program the indirect write of the DCC flags occurs immediately after the direct DTR access.

  Note

  For an access to DBGDTR_EL0, this means the indirect write happens after both DBGDTRRX_EL0 and DBGDTRTX_EL0 have been accessed.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- In the simple sequential execution model, for an external read of DBGDTRTX_EL0 or an external write of DBGDTRRX_EL0:
  - The check of the DCC flags for overrun or underrun occurs immediately before the access.
  - If there is no underrun or overrun, the update of the DCC flags occurs immediately after the access.
  - If there is underrun or overrun, the update of the DCC underrun or overrun flags occurs immediately after the access.

All observers must observe the same order for accesses.

Note

These requirements do not create order where order does not otherwise exist. It applies only for ordered accesses.

Without explicit synchronization following external writes and external reads:

- The value written by the external write to DBGDTRRX_EL0 that does not overrun, must be observable to direct reads of DBGDTRRX_EL0, Debug Data Transfer Register, Receive and DBGDTR_EL0 in finite time.
- The DCC flags that are updated as a side-effect of the external write or external read must be observable:
  - To subsequent external reads of EDSCR.
  - To subsequent external reads of DBGDTRRX_EL0 when checking for underrun.
  - To subsequent external writes to DBGDTRTX_EL0 when checking for overrun.
  - To direct reads of MDCCSR_EL0, Monitor DCC Status Register in finite time.

However, explicit synchronization is required to guarantee that a direct read of MDCCSR_EL0, Monitor DCC Status Register returns up-to-date DCC flags. This means that if a signal is received from another agent that indicates that MDCCSR_EL0, Monitor DCC Status Register must be read, an `ISB` is required to ensure that the direct read of MDCCSR_EL0, Monitor DCC Status Register occurs after the signal has been received. This also synchronizes the value in DBGDTRRX_EL0, Debug Data Transfer Register, Receive, if applicable. However, if that signal is an interrupt exception triggered by **COMMIRQ**, **COMMTX**, or **COMMRX**, the exception entry is sufficient synchronization. See Synchronization of DCC interrupt request signals.

Explicit synchronization is required following a direct read or direct write:

- To ensure that a value directly written to DBGDTRTX_EL0, Debug Data Transfer Register, Transmit is observable to external reads of DBGDTRTX_EL0.
- To ensure that a value directly written to DBGDTR_EL0 is observable to external reads of DBGDTRTX_EL0 and DBGDTRRX_EL0.
- To guarantee that the indirect writes to the DCC flags that were a side-effect of the direct read or direct write have occurred, and therefore that the updated values are:
  - Observable to external reads of EDSCR.
  - Observable to external reads of DBGDTRRX_EL0 when checking for underrun.
  - Observable to external writes of DBGDTRTX_EL0 when checking for overrun.
  - Returned by a following direct read of MDCCSR_EL0, Monitor DCC Status Register.

See also Memory-mapped accesses to the DCC and ITR and Synchronization of changes to the external debug registers.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

These ordering rules mean that software:

- Must not read DBGDTRRX_EL0, Debug Data Transfer Register, Receive without first checking MDCCSR_EL0, Monitor DCC Status Register.RXfull or if the previously-read value of MDCCSR_EL0, Monitor DCC Status Register.RXfull is 0.

  It is not sufficient to read both registers and then later decide whether to discard the read value, as there might be an intervening write from the external debug interface.

- Must not write DBGDTRTX_EL0, Debug Data Transfer Register, Transmit without first checking MDCCSR_EL0, Monitor DCC Status Register.TXfull or if the previously-read value of MDCCSR_EL0, Monitor DCC Status Register.TXfull is 1.

  The write to DBGDTRTX_EL0, Debug Data Transfer Register, Transmit overwrites the value in DTRTX, and the external debugger might or might not have read this value.

- Must ensure there is an explicit Context Synchronization event following a DTR access, even if not immediately returning to read MDCCSR_EL0, Monitor DCC Status Register again. This synchronization operation can be an exception return.

E4.5.2.1 Derived requirements

The rules for DCC accesses in Non-debug state are as follows:

- Following a direct read of DBGDTRRX_EL0, Debug Data Transfer Register, Receive when RXfull is 1:
  - If an external write to DBGDTRRX_EL0, Debug Data Transfer Register, Receive checks the RXfull flag for overrun and observes that the value of RXfull is 0, the value returned by the previous direct read must not be affected by the external write.
  - If an external read of EDSCR returns a RXfull value of 0, then the value returned by the previous direct read must not be affected by a following external write to DBGDTRRX_EL0, Debug Data Transfer Register, Receive, and the following external write does not overrun.
- Following a direct read of DBGDTR_EL0, when RXfull is 1:
  - If an external write to DBGDTRRX_EL0, Debug Data Transfer Register, Receive checks the RXfull flag for overrun and observes that the value of RXfull is 0, the value returned by the previous direct read must not be affected by the external write nor by a following direct write to DBGDTRTX_EL0, Debug Data Transfer Register, Transmit.
  - If an external read of EDSCR returns a RXfull value of 0, then the value returned by the previous direct read must not be affected by subsequent external writes to DBGDTRRX_EL0, Debug Data Transfer Register, Receive and DBGDTRTX_EL0, Debug Data Transfer Register, Transmit in any order, and the following external write of DBGDTRRX_EL0, Debug Data Transfer Register, Receive will not overrun.
- Following a direct write to DBGDTRTX_EL0, Debug Data Transfer Register, Transmit, when TXfull is 0:
  - If an external read of DBGDTRTX_EL0, Debug Data Transfer Register, Transmit checks the TXfull flag for underrun and observes that the value of TXfull is 1, the value returned by the external read must be the value written by the previous direct write.
  - If an external read of EDSCR returns a TXfull value of 1, then the value returned by a following external read of DBGDTRRX_EL0, Debug Data Transfer Register, Receive must be the value written by the previous direct read, and the subsequent external read will not underrun.
- Following a direct write to DBGDTR_EL0, when TXfull is 0:
  - If an external read of DBGDTRTX_EL0, Debug Data Transfer Register, Transmit checks the TXfull flag for underrun and observes that the value of TXfull is 1, the values returned by the external read and by a subsequent external read of DBGDTRRX_EL0, Debug Data Transfer Register, Receive must be the value written by the previous direct write.
  - If an external read of EDSCR returns a TXfull value of 1, then the value returned by subsequent external reads of DBGDTRRX_EL0, Debug Data Transfer Register, Receive and DBGDTRTX_EL0, Debug Data Transfer Register, Transmit, in any order, must be the value written by the previous direct read, and the subsequent external read of DBGDTRTX_EL0, Debug Data Transfer Register, Transmit does not underrun.
- Following an external read of DBGDTRTX_EL0, Debug Data Transfer Register, Transmit that does not underrun, if a direct read of MDCCSR_EL0, Monitor DCC Status Register returns a TXfull value of 0, then the

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

value returned by the external read must not be affected by a following direct write to DBGDTRTX_EL0, Debug Data Transfer Register, Transmit.

- Following a first external read DBGDTRRX_EL0, Debug Data Transfer Register, Receive and a following second external read of DBGDTRTX_EL0, Debug Data Transfer Register, Transmit that does not underrun, if a direct read of MDCCSR_EL0, Monitor DCC Status Register returns a TXfull value of 0, then the values returned by the external reads must not be affected by a following direct write to DBGDTR_EL0.
- Following an external write to DBGDTRRX_EL0, Debug Data Transfer Register, Receive that does not overrun, if a direct read of MDCCSR_EL0, Monitor DCC Status Register returns an RXfull value of 1, then the value returned by a following direct read of DBGDTRRX_EL0, Debug Data Transfer Register, Receive or DBGDTR_EL0 must be the value written by the previous external write.
- Following a first external write to DBGDTRTX_EL0, Debug Data Transfer Register, Transmit and a following second external write to DBGDTRRX_EL0, Debug Data Transfer Register, Receive that does not overrun, if a direct read of MDCCSR_EL0, Monitor DCC Status Register returns an RXfull value of 1, then the value returned by a subsequent direct read of DBGDTR_EL0 must return the values written by the previous external writes.

E4.5.3 Synchronization of DCC interrupt request signals

Following an external read or external write access to the DTR registers, the interrupt request signals, **COMMIRQ**, **COMMTX**, and **COMMRX**, must be updated in finite time without explicit synchronization.

The updated values must be observable to a direct read of MDCCSR_EL0, Monitor DCC Status Register or DBGDTRRX_EL0, Debug Data Transfer Register, Receive, or a direct write of DBGDTRTX_EL0, Debug Data Transfer Register, Transmit executed after taking an interrupt exception generated by the interrupt request. The updated values must also be observable to a direct write of DBGDTRTX_EL0, Debug Data Transfer Register, Transmit executed after taking an interrupt exception generated by the interrupt request.

Note

The requirement that indirect writes to registers are observable to direct reads in finite time does not imply that all observers will observe the indirect write at the same time. For more information, see Synchronization requirements for AArch64 System registers.

Following a direct read of DBGDTRRX_EL0, Debug Data Transfer Register, Receive or a direct write to DBGDTRRX_EL0, Debug Data Transfer Register, Receive, software must execute a Context Synchronization event to guarantee the interrupt request signals have been updated in finite time. This synchronization operation can be an exception return.

E4.5.4 DCC and ITR access in Debug state

In Debug state, stricter observability rules apply for instructions issued through the ITR, to maintain communication between a debugger and the PE, without requiring excessive explicit synchronization.

In Normal access mode, without explicit synchronization:

- A direct read or direct write of the DTR registers by an instruction written to EDITR must be observable to an external write or an external read in finite time:
  - A direct read of DBGDTRRX_EL0, Debug Data Transfer Register, Receive must be observable to an external write of DBGDTRRX_EL0.
  - A direct read of DBGDTR_EL0 must be observable to an external write of DBGDTRRX_EL0 and DBGDTRTX_EL0.
  - A direct write of DBGDTRTX_EL0, Debug Data Transfer Register, Transmit must be observable to an external read of DBGDTRTX_EL0.
  - A direct write of DBGDTR_EL0 must be observable to an external read of DBGDTRRX_EL0 and DBGDTRTX_EL0.

  This includes the indirect write to the DCC flags that occurs atomically with the access as described in DCC accesses in Non-debug state.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The subsequent external write or external read must observe either the old or the new values of both the DTR contents and DCC flags. If the old values are observed, this typically results in overrun or underrun, assuming the old values of the DCC flags indicate an overrun or underrun condition, as would normally be the case.

This means the debugger can observe the direct read or direct write without explicit synchronization and without explicitly testing the DCC flags in EDSCR, because it can rely on overrun and underrun tests.

- External reads of DBGDTRTX_EL0 that do not underrun and external writes to DBGDTRRX_EL0 that do not overrun must be observable to an instruction subsequently written to EDITR on completion of the first external access. This includes the indirect write to the DCC flags.

  This means that without explicit synchronization and without the need to first check the DCC flags in MDCCSR_EL0, Monitor DCC Status Register:

  — If the instruction is a direct read of DBGDTRRX_EL0, Debug Data Transfer Register, Receive, it observes the external write.

  — If the instruction is a direct write of DBGDTRTX_EL0, Debug Data Transfer Register, Transmit, it observes the external read.

- Writes to EDITR that do not overrun commit an instruction for execution immediately. The instruction must complete execution in finite time without requiring any further operation by the debugger.
- After an external write to the EDITR, the ITR flags that are updated as a side effect of that write must be observable by:

  — An external read of the EDSCR that follows the external write to the EDITR.

  — When checking for overrun, another external write to the EDITR that follows the original external write to the EDITR.

In Memory access mode, these requirements shift to the instructions implicitly executed by external reads and external writes of the DTR registers, as described in Memory access mode.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E4.6 Interrupt-driven use of the DCC

Arm recommends implementations provide a level-sensitive DCC interrupt request through the IMPLEMENTATION DEFINED interrupt controller as a Private Peripheral Interrupt for the originating PE.

Note

- In addition to connection to the interrupt controller Arm also recommends **COMMIRQ**, **COMMTX**, and **COMMRX** signals that might be implemented for use by any legacy system peripherals.
- GICv3 reserves a Private Peripheral Interrupt number for the **COMMIRQ** interrupt.

The MDCCINT_EL1, Monitor DCC Interrupt Enable Register register provides a first level of interrupt masking within the PE, meaning only a single interrupt source, **COMMIRQ**, is needed at the interrupt controller.

See also Synchronization of DCC interrupt request signals.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E4.7 Pseudocode description of the operation of the DCC and ITR registers

The basic operation of the DCC and ITR registers is shown by the following pseudocode functions. These functions do not cover the behavior when OSLSR_EL1, OS Lock Status Register.OSLK == 1, meaning that the OS Lock is locked:

- `Write_DBGDTR_EL0[].`
- `Write_DBGDTRRX_EL0[].`
- `Write_DBGDTRTX_EL0[].`
- `Write_EDITR[].`
- `CheckForDCCInterrupts().`

For the definition of the DTR Registers, see shared/debug/dccanditr/DTR.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter E5
The Embedded Cross-Trigger Interface

This chapter describes the Embedded Cross-Trigger interface. It contains the following sections:

- About the Embedded Cross-Trigger.
- Basic operation on the ECT.
- Cross-triggers on a PE in an Arm A-profile implementation.
- Description and allocation of CTI triggers.
- CTI registers programmers' model.
- Examples.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E5.1 About the Embedded Cross-Trigger

The *Embedded Cross-Trigger* (ECT) allows a debugger to:

- Send trigger events to a PE. For example, this might be done to halt the PE.
- Send a trigger event to one or more PEs, or other system components, when a trigger event occurs on another PE or system component. For example, this might be done to halt all PEs when one individual PE halts.

Figure E5-1 shows the logical structure of an ECT.

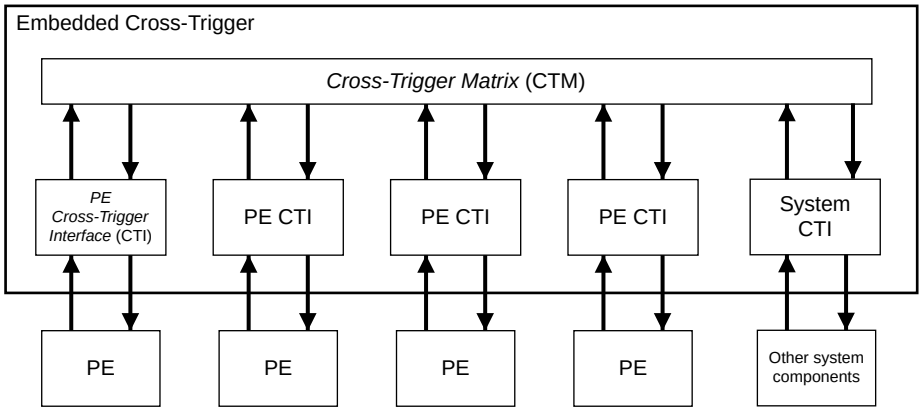


Figure E5-1 Structure of an Embedded Cross-Trigger

The ECT can deliver many types of trigger events, which are described in the following sections:

- Debug request trigger event.
- Restart request trigger event.
- Cross-halt trigger event.
- Performance Monitors overflow trigger event.
- Generic trace external input trigger events.
- Generic trace external output trigger events.
- Generic CTI interrupt trigger event.

From the introduction of Armv8, an A-profileR-profile implementation must:

- Include a Cross-Trigger interface, CTI.
- Implement at least the input and output triggers defined in this architecture.

In addition, see Cross-triggers on a PE in an Arm R-profile implementation.

Arm recommends that this Cross-Trigger interface includes:

- The ability to route trigger events between Trace Units, which typically have advanced event triggering logic.
- An output trigger to the interrupt controller.

Also, Arm recommends that the Embedded Cross-Trigger includes the capability to send and receive IMPLEMENTATION DEFINED system trigger events to and from other system components, including a system counter, using a system CTI. See Halt-on-debug.

Note

The ECT and CTI must only signal trigger events for external debugging. They must not route software events, such as interrupts. For example, the Performance Monitors overflow input trigger is provided to allow entry to Debug state on a counter overflow, and the output trigger to the interrupt controller is provided to generally allow events from the external debug subsystem to be routed to a software agent. However, the combination of the two must not be used as a mechanism to route Performance Monitors overflows to an interrupt controller.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

CTI version 1 (CTIv1) is defined by the *CoreSight™ SoC Technical Reference Manual*. CTIv2 extends CTIv1 with the addition of the input channel gate. See Implementation with CTIv2.

E5.1.1 Implementation with a CoreSight CTI

For details of the recommended connections for an A-profile implementation, see Recommended External Debug Interface. See also *CoreSight™ SoC Technical Reference Manual.*

E5.1.2 Implementation with CTIv2

If the CTI implemented is CTIv2 then:

- The CTIDEVARCH, CTIDEVAFF0, and CTIDEVAFF1 registers must be implemented.
- If the channel gate function is implemented, it applies to both input and output channels.
- The input channel gate function must be implemented if either of the following is true:
  - The CTM is implemented and the architecture variant is Armv8.5 or higher.
  - The CTIDEVARCH.REVISION field reads as `0b0001` or higher.

Implementation of CTIv2 features in architecture variants below Armv8.5 is OPTIONAL, but Arm recommends that CTIv2 is implemented, CTIv2 must be implemented from Armv8.5.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E5.2 Basic operation on the ECT

The ECT comprises a Cross-Trigger Matrix, CTM, and one Cross-Trigger Interface, CTI, for each PE. The ECT might also include other CTIs for other system components. The CTM passes events between the CTI blocks over channels. The CTM can have a maximum of 32 channels.

The main interfaces of the Cross-Trigger interface, CTI, are:

- The input triggers:
  - These are trigger event inputs from the PE to the CTI.
- The output triggers:
  - These are trigger event outputs from the CTI to the PE.
- The input channels:
  - These are channel event inputs from the Cross-Trigger matrix, CTM, to the CTI.
- The output channels:
  - These are channel event outputs from the CTI to the CTM.

Each CTI block has:

- Up to 32 input triggers that come from the PE:
  - The input triggers are numbered 0-31.
- Up to 32 output triggers that go to the PE:
  - The output triggers are numbered 0-31.

If the CTI is not powered up when the Core power domain is powered up, the CTI ignores all input triggers and input channel events, and does not generate any output triggers or output channel events.

Figure E5-2 shows the logical internal structure of a CTI.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

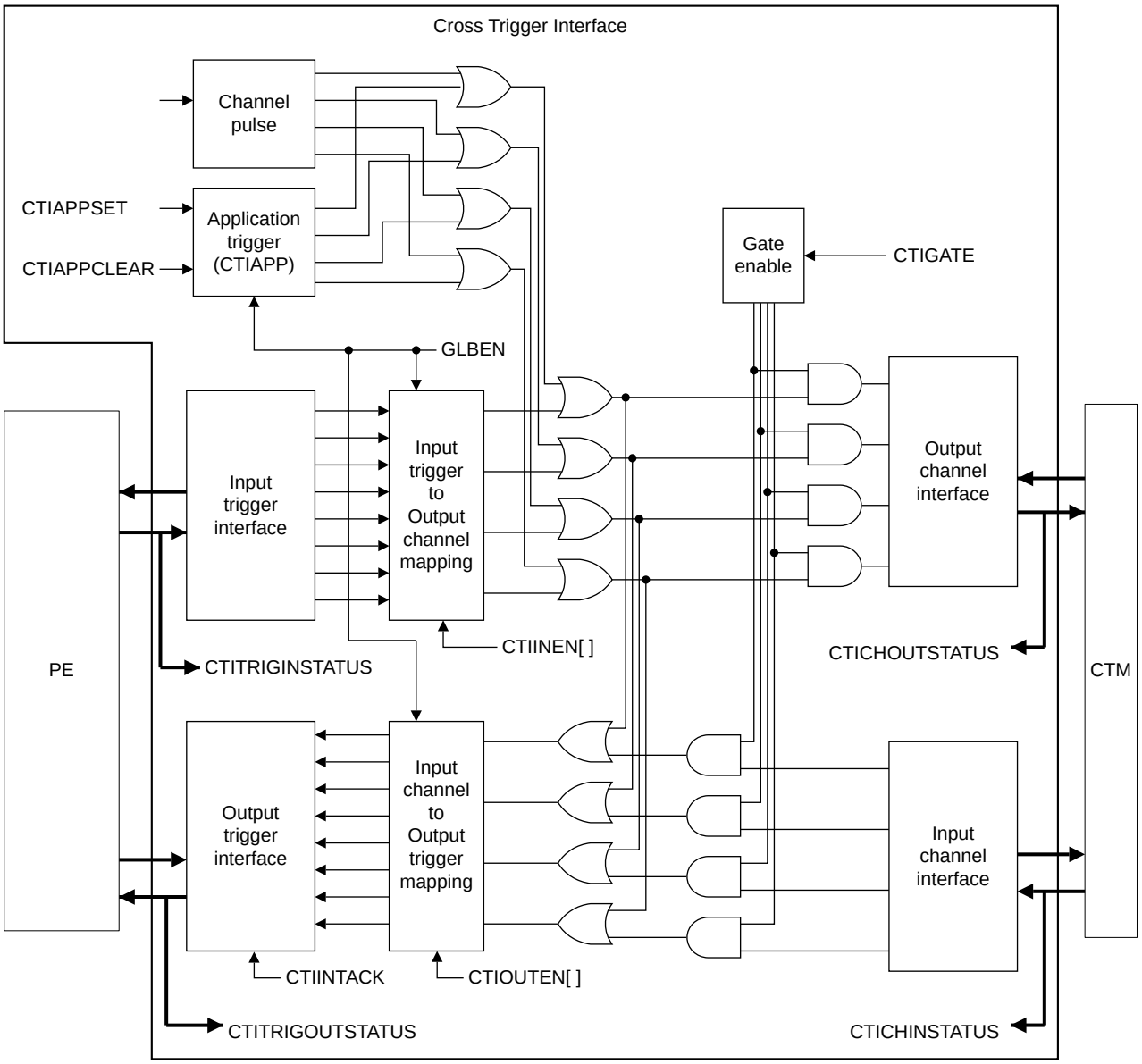


Figure E5-2 Structure of a Cross-Trigger interface

Note

- The number of triggers in IMPLEMENTATION DEFINED. Figure E5-2 shows eight input and eight output triggers.
- The number of channels is IMPLEMENTATION DEFINED. Figure E5-2 shows four channels.
- In Figure E5-2 the input channel gate function is a CTIv2 feature.

When the CTI receives an input trigger event, this generates channel events on one or more internal channels, according to the *mapping function* defined by the *Input trigger→output channel mapping registers*, CTIINEN<n>.

The CTI also contains an *application trigger* and *channel pulse* to allow a debugger to create channel events directly on internal channels by writing to the CTI control registers.

Channel events on each internal channel are passed to a corresponding output channel that is controlled by a *channel gate*. The channel gate can block propagation of channel events from an internal channel to an output channel.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

If the CTM is implemented:

- The gate function must be implemented.
- If the CTI is CTIv1, the gate function applies to output triggers only.

The output channels from a CTI are combined, using a logical OR function, with the output channels from all other CTIs to form the input channels on other CTIs. The input channels of this CTI are the logical OR of the output channels on all other CTIs. This is the *Cross-Trigger Matrix*, CTM. Therefore, the number of input channels must equal the number of output channels.

Note

The number of input triggers and output triggers is not required to be the same.

The internal channels form an internal Cross-Trigger matrix within the CTI. This delivers events directly from the input triggers to the output triggers. Therefore the number of internal channels is the same as the number of input and output channels on the external CTM, and there is a direct mapping between the two.

Channel events received on each input channel are passed to the corresponding internal channel. It is IMPLEMENTATION DEFINED whether the Cross-Trigger gate also blocks propagation of channel events from input channels to internal channels.

Note

If CTIv2 is implemented, the Cross-Trigger gate also blocks propagation of channel events from input channels to internal channels.

When the CTI receives a channel event on an internal channel this generates trigger events on one or more output triggers, according to the mapping function defined by the *Input channel → output trigger mapping registers*, CTIOUTEN<n>.

The CTI contains the input and output trigger interfaces to the PE and the interface of the Cross-Trigger matrix. The architecture does not define the signal protocol used on the trigger interfaces, and:

- It is IMPLEMENTATION DEFINED whether the CTI supports multicycle input trigger events.
- It is IMPLEMENTATION DEFINED whether the CTM supports multicycle channel events.

See Multicycle events.

However, an output trigger is asserted until acknowledged. The output trigger can be:

- Self-acknowledging. This means that no further action is required from the debugger.
- Acknowledged by the debugger writing 1 to the corresponding bit of CTIINTACK.

The time taken to propagate a trigger event from the first PE, through its CTI, across the CTM to another CTI, and thereby to a second PE is IMPLEMENTATION DEFINED.

Note

Arm recommends that this path is not longer than the shortest software communication path between those PEs. This is because if the first PE halts, the Cross-halt trigger event can propagate through the ECT and halt the second PE without causing software on the second PE to malfunction because the first PE is in Debug state and is not responding.

E5.2.1 Multicycle events

A multicycle event is one with a continuous state that might persist over many cycles, as opposed to a discrete event. A typical implementation of a multicycle event is a level-based signal interface, whereas a discrete event might be implemented as a pulse signal or message.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CTI support for multicycle trigger events is IMPLEMENTATION DEFINED. Use of multicycle trigger events is deprecated. Of the architecturally defined input trigger events, the Performance Monitors overflow trigger event and Generic trace external output trigger events can be multicycle input triggers.

CTM support for multicycle channel events is IMPLEMENTATION DEFINED. A CTM that does not support multicycle channel events cannot propagate a multicycle trigger event between CTIs.

Note

A full ECT might comprise a mix of CTIs, some of which can support multicycle trigger events. In bridging these components, multicycle channel events become single channel events at the boundary between the CTIs.

E5.2.1.1 An ECT that supports multicycle trigger events

When an ECT supports multicycle trigger events, an input trigger event to the CTI continuously asserts channel events on all output channels mapped to it until either:

- The input trigger event is removed.
- The channel mapping function is disabled.

This means that an input trigger that is asserted for multiple cycles causes any channels that are mapped to it to become active for multiple cycles. Consequently, any output triggers mapped from that channel are asserted for multiple cycles.

Note

The output trigger remains asserted for at least as long as the channel remains active. This means that even if the output trigger is acknowledged, it remains asserted until the channel deactivates.

The CTI does not guarantee that these events have precisely the same duration, as the triggers and channels can cross between clock domains.

CTIAPPSET and CTIAPPCLEAR can set a channel active for multiple cycles. CTIAPPPULSE generates a single channel event. CTICHINSTATUS and CTICHOUTSTATUS can report whether a channel is active.

E5.2.1.2 An ECT that does not support multicycle trigger events

When an ECT does not support multicycle trigger events, an input trigger event to the CTI generates a single channel event on all output channels mapped to it, regardless of how long the input trigger event is asserted.

This means that an input trigger event that is asserted for multiple cycles generates a single channel event on any channels mapped to it. Consequently any self-acknowledging output triggers mapped from those channels are single trigger events.

Note

A single event is typically a single cycle, but there is no guarantee that this is always the case.

CTIAPPSET and CTIAPPCLEAR can only generate a single channel event. CTIAPPPULSE generates a single channel event. If the ECT does not support multicycle channel events, use of CTIAPPSET and CTIAPPCLEAR is deprecated, and the debugger must only use CTIAPPPULSE. CTICHINSTATUS and CTICHOUTSTATUS must be treated as UNKNOWN.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E5.3 Cross-triggers on a PE in an Arm R-profile implementation

An R-profile PE must include a Cross-Trigger interface, and the implementation must include at least the input and output triggers defined in this architecture. The number of channels in the Cross-Trigger matrix is IMPLEMENTATION DEFINED, but there must be a minimum of three. Software can read CTIDEVID.NUMCHAN to discover the number of implemented channels.

The CTM must connect to all PEs in the same Inner Shareability domain as the PE that includes the Cross-Trigger interface, but can also connect to additional PEs. Arm strongly recommends that the CTM connects all PEs implementing a CTI in the system. This includes other PEs that can be connected using a CoreSight CTI module.

Note

In a uniprocessor system the CTM is OPTIONAL. In a multiprocessor system the CTM is required. The CTM might be connected other CTI modules for non-PEs, such as triggers for system visibility components. Arm recommends that the CTM is implemented.

Any CTI connected to a PE that is not an Arm PE must implement at least:

- The Debug request trigger event.
- The Restart trigger event.
- The Cross-halt trigger event.

For more information about the CTI, see the *CoreSight™ SoC Technical Reference Manual*. Arm architecture refines the generic CTI by defining roles for each of the implemented input and output triggers.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E5.4 Description and allocation of CTI triggers

Table E5-1 shows the output trigger events defined by the architecture and the related trigger numbers.

Table E5-1 Allocation of CTI output trigger events

Number	Source	Destination	Event description
0	CTI	PE	Debug request trigger event
1	CTI	PE	Restart request trigger event
2	CTI	IRQ controller	Generic CTI interrupt trigger event
3	-	-	Reserved
4 - 7	CTI	Trace unit	OPTIONAL Generic trace external input trigger events
8 - 15	-	-	Reserved

Note

Output triggers from the CTI are inputs to other blocks.

Table E5-2 shows the input trigger events defined by the architecture and the related trigger numbers.

Table E5-2 Allocation of CTI input trigger events

Number	Source	Destination	Event description
0	PE	CTI	Cross-halt trigger event
1	PE	CTI	Performance Monitors overflow trigger event
2	-	-	Reserved
3	-	-	Reserved
4 - 7	Trace unit	CTI	OPTIONAL Generic trace external output trigger events
8 - 10	-	-	Reserved
11 - 15	-	-	IMPLEMENTATION DEFINED.

Note

Input triggers to the CTI are outputs from other blocks.

Table E5-1 and Table E5-2 show the minimum set of trigger events defined by the architecture. However:

- The Generic trace external input and output trigger events are required only if the OPTIONAL trace unit is implemented. If the OPTIONAL trace unit is not implemented, these trigger events are reserved.
- Support for the generic CTI interrupt trigger event is IMPLEMENTATION DEFINED because details of interrupt handling in the system, including any interrupt controllers, are IMPLEMENTATION DEFINED. Details regarding how the CTI interrupt is connected to an interrupt controller and its allocated interrupt number lie outside the scope of the architecture. Arm strongly recommends that implementations provide a means to generate interrupts based on external debug events.
- The other trigger events are required by the architecture.

An implementation can extend the CTI with additional triggers.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E5.4.1 Debug request trigger event

This is an output trigger event from the CTI, and an input trigger event to the PE, asserted by the CTI to force the PE into Debug state. The trigger event is asserted until acknowledged by the debugger. The debugger acknowledges the trigger event by writing 1 to CTIINTACK[0].

Note

A debugger must poll CTITRIGOUTSTATUS[0] until it reads as 0, to confirm that the output trigger has been deasserted before generating any event that must be ordered after the write to CTIINTACK, such as a write to CTIAPPPULSE to activate another trigger.

If the PE is already in Debug state, the PE ignores the trigger event, but the CTI continues to assert it until it is removed by the debugger. See also External Debug Request debug event.

E5.4.2 Restart request trigger event

This is an output trigger event from the CTI, and an input trigger event to the PE, asserted by the CTI to request the PE to exit Debug state. If the PE is in Non-debug state, the request is ignored by the PE.

If a Restart request trigger event is received at or about the same time as the PE enters Debug state, it is CONSTRAINED UNPREDICTABLE whether:

- The request is ignored by the PE. In this case the PE enters Debug state and remains in Debug state.
- The PE enters Debug state and then immediately restarts.

Debuggers must program the CTI to send Restart request trigger events only to PEs that are halted. To enable the PE to disambiguate discrete Restart request trigger events, after sending a Restart request trigger event, the debugger must confirm that the PE has restarted and halted before sending another Restart request trigger event. Debuggers can use EDPRSR.{SDR, HALTED} to determine the Execution state of the PE.

Note

Before generating a Restart request trigger event for a PE, a debugger must ensure any Debug request trigger event targeting that PE is cleared. Debug request trigger event describes how to do this.

The trigger event is self-acknowledging, meaning that the debugger requires no further action to remove the trigger event. The trigger event is acknowledged even if the request is ignored by the PE. See also Exiting Debug state.

E5.4.3 Cross-halt trigger event

This is an input trigger event to the CTI, and an output trigger event from the PE, asserted by a PE when it is entering Debug state.

Note

To reduce the latency of halting, Arm recommends that an implementation issues the Cross-halt trigger event early in the committed process of entering Debug state. This means that there is no requirement to wait until all aspects of entry to Debug state have completed before issuing the trigger event. Speculative emission of Cross-halt trigger events is not allowed. The Cross-halt trigger event must not be issued early enough for a subsequent Debug request trigger event, which might be derived from the Cross-halt trigger event, to be recorded in the EDSCR.STATUS field. This applies to Debug request trigger events that are acting as inputs to the PE.

E5.4.4 Performance Monitors overflow trigger event

This is an input trigger event to the CTI, and an output trigger event from the PE, asserted each time the PE records a Performance Monitors counter overflow. See The Performance Monitors Extension.

If the CTI supports multicycle trigger events, then the trigger event remains asserted while any of the following are true, and deasserted otherwise:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- An event counter <n> is implemented, the global enable control for the event counter is 1, PMINTENSET_EL1, Performance Monitors Interrupt Enable Set Register.P[n] is 1, and PMOVSSET_EL0, Performance Monitors Overflow Flag Status Set Register.P[n] is 1. The global enable control is defined in Enabling PMU counters.
- [**multi:pmctr?**].E is 1, PMINTENSET_EL1, Performance Monitors Interrupt Enable Set Register.C is 1, and PMOVSSET_EL0, Performance Monitors Overflow Flag Status Set Register.C is 1.

Note

- This does not replace the recommended connection of Performance Monitors overflow trigger event to an interrupt controller. Software must be able to program an interrupt on Performance Monitors overflow without programming the CTI.
- Events can be counted when `ExternalNoninvasiveDebugEnabled()` == FALSE, and, in Secure state, when `ExternalSecureNoninvasiveDebugEnabled()` == FALSE. Secure software must be aware that overflow trigger events are nevertheless visible to the CTI.

If the CTI does not support multicycle trigger events, then the trigger event is asserted whenever a counter overflows, causing the PE to set PMOVSSET_EL0, Performance Monitors Overflow Flag Status Set Register.P[n] to 1, or a direct write to a System register means the above conditions become true.

For more information, see An ECT that supports multicycle trigger events.

E5.4.5 Generic trace external input trigger events

These are output trigger events from the CTI, and input trigger events to the OPTIONAL trace unit, that are used in conjunction with the Generic trace external output trigger events to pass trigger events between:

- The PE and the OPTIONAL trace unit.
- The OPTIONAL trace unit and any other component attached to the CTM, including other Trace Units.

There are four Generic trace external input trigger events.

The trigger events are self-acknowledging. This means that the debugger does not have to take any further action to remove the events.

E5.4.6 Generic trace external output trigger events

These are input trigger events to the CTI, and output trigger events from the OPTIONAL trace unit, used in conjunction with the Generic trace external input trigger events to pass trigger events between:

- The PE and the OPTIONAL trace unit.
- The OPTIONAL trace unit and any other component attached to the CTM, including other Trace Units.

There are four Generic trace external output trigger events.

E5.4.7 Generic CTI interrupt trigger event

This is an output trigger event from the CTI, and an input to an IMPLEMENTATION DEFINED interrupt controller, and can transfer trigger events from the PE, trace units, or any other component attached to the CTI and CTM to software as an interrupt. The Generic CTI interrupt trigger event must be connected to the interrupt controller as an interrupt that can target the originating PE.

Note

- Arm recommends that the Generic CTI interrupt trigger event is a private peripheral interrupt, but implementations might instead make this trigger event available as a shared peripheral interrupt or a local peripheral interrupt.
- GICv3 reserves a private peripheral interrupt number for this interrupt.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

It is IMPLEMENTATION DEFINED whether this trigger event is:

- Self-acknowledging. This means that the debugger is not required to take any further action, and that the interrupt controller must treat the trigger event as a pulse or edge-sensitive interrupt.
- Acknowledged by the debugger. The debugger acknowledges the trigger event by writing 1 to CTIINTACK[2]. This means that the interrupt controller must treat the trigger event as a level-sensitive interrupt.

Arm recommends that the Generic CTI interrupt trigger event is a self-acknowledging trigger event.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E5.5 CTI registers programmers' model

The CTI registers programmers' model is described in About the External Debug Registers. The following sections contain information specific to the CTI:

- External debug register resets.
- External debug interface register access permissions.
- Cross-trigger interface registers.
- The individual register descriptions in Cross-Trigger Interface registers.

See also Memory-mapped accesses to the external debug interface.

E5.5.1 CTI reset

An External Debug reset resets the CTI. See External debug register resets for details of CTI register resets. All CTI output triggers and output channels are deasserted on an External Debug reset.

Note

An indirect read of an output trigger might not observe the deasserted state until the processor is Cold reset. For more information, see Synchronization of changes to the external debug registers.

E5.5.2 CTI authentication

The CTI ignores the state of the IMPLEMENTATION DEFINED authentication interface. This means that:

- CTITRIGINSTATUS shows the status of the input triggers and CTICHINSTATUS shows the status of the input channels, regardless of the value of `ExternalNoninvasiveDebugEnabled()`.

  Note

  The PE does not generate the Cross-halt trigger event and the trace unit does not generate Generic trace external output trigger events when `ExternalNoninvasiveDebugEnabled()` == FALSE. However, the PE can generate Performance Monitors overflow trigger events.

- The CTI can generate external triggers regardless of the value of `ExternalInvasiveDebugEnabled()`.

  Note

  The PE ignores Debug request and Restart request trigger events when `ExternalInvasiveDebugEnabled()` is FALSE.
  The trace unit ignores Generic trace external input trigger events when `ExternalNoninvasiveDebugEnabled()` is FALSE.
  The behavior of Generic CTI interrupt requests is part of the IMPLEMENTATION DEFINED handling of these interrupts, but it is permissible for an interrupt controller to receive these requests even when `ExternalInvasiveDebugEnabled()` is FALSE.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E5.6 Examples

The CTI is fully programmable and allows for flexible cross-triggering of events within a PE and between PEs in a multiprocessor system. For example:

- The Cross-halt trigger event and the Debug request trigger event can be used for cross-triggering in a multiprocessor system.
- The Cross-halt trigger event and the Generic interrupt trigger event can be used for event-driven debugging in a multiprocessor system.
- The Performance Monitors overflow trigger event and the Debug request trigger event can force entry to Debug state on overflow of a Performance Monitors event counter, for event-driven profiling.

  Note

  This does not replace the recommended connection of Performance Monitors overflow trigger events to an interrupt controller. Software must be able to program an interrupt on Performance Monitors overflow without programming the CTI. Arm recommends that the Performance Monitors overflow signal is directly available as a local interrupt source.

- The Generic trace external input and Generic trace external output trigger events can pass trace events into and out of the event logic of the trace unit. They can do this:
  - To pass trace events between Trace Units.
  - In conjunction with the Performance Monitors overflow trigger event, to couple the Performance Monitors to the PE trace unit.
  - In conjunction with the Debug request trigger event, to trigger entry to Debug state on a trace event.
  - In conjunction with other CTIs, to signal a trace trigger event onto a CoreSight trace interconnect.

The following sections describe some examples in more detail:

- Example E5-1.
- Example E5-2.
- Example E5-3.
- Example E5-4.

Example E5-1 Halting a single PE

To halt a single PE, set:

1. CTIGATE[0] to 0, so that the CTI does not pass channel events on internal channel 0 to the CTM.
2. CTIOUTEN0[0] to 1, so that the CTI generates a Debug request trigger event in response to a channel event on channel 0.

   Note

   The Cross-halt trigger event is input trigger 0, meaning it is controlled by the instance of CTIOUTEN<n> for which <n> is 0.

3. CTIAPPPULSE[0] to 1, to generate a channel event on channel 0.

When the PE has entered Debug state, clear the Debug request trigger event by writing 1 to CTIINTACK[0], before restarting the PE.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example E5-2 Halting all PEs in a group when any one PE halts

To program a group of PEs so that when one PE in the group halts, all of the PEs in that group halt, set the following registers for each PE in the group:

1. CTIGATE[2] to 1, so that each CTI passes channel events on internal channel 2 to the CTM.
2. CTIINEN0[2] to 1, so that each CTI generates a channel event on channel 2 in response to a Cross-halt trigger event.
3. CTIOUTEN0[2] to 1, so that each CTI generates a Debug request trigger event in response to a channel event on channel 2.

   Note

   The Cross-halt trigger event is input trigger 0, meaning it is controlled by the instances of CTIINEN<n> and CTIOUTEN<n> for which <n> is 0.

When a PE has halted, clear the Debug request trigger event by writing a value of 1to CTIINTACK[0], before restarting the PE.

Example E5-3 Synchronously restarting a group of PEs

To restart a group of PEs, for each PE in the group:

1. If the PE was halted because of a Debug request trigger event, the debugger must ensure the trigger event is deasserted. It can do this by:
   a. Writing 1 to CTIINTACK[0] to clear the Debug request trigger event.
   b. Polling CTITRIGOUTSTATUS[0], until it reads as 0, to confirm that the trigger event has been deasserted.
2. Set CTIGATE[1] to 1, so that each CTI passes channel events on internal channel 1 to the CTM.
3. Set CTIOUTEN1[1] to 1, so that each CTI generates a Restart request trigger event in response to a channel event on channel 1.

   Note

   This example must use the instance of CTIOUTEN<n> for which <n> is 1.

4. Set CTIAPPPULSE[1] to 1 on any one PE in the group, to generate a channel event on channel 1.

Example E5-4 Halting a single PE on Performance Monitors overflow

To halt a single PE on a Performance Monitors overflow set:

1. CTIGATE[3] to 0, so that the CTI does not pass channel events on internal channel 3 to the CTM.
2. CTIINEN1[3] to 1, so that the CTI generates a channel event on channel 3 in response to a Performance Monitors overflow trigger event.

   Note

   This step of this example must use the instance of CTIINEN<n> for which <n> is 1.

3. CTIOUTEN0[3] to 1, so that the CTI generates a Debug request trigger event in response to a channel event on channel 3.

   Note

   This step of this example must use the instance of CTIOUTEN<n> for which <n> is 0.

When the PE has entered Debug state, clear the Debug request trigger event by writing 1 to CTIINTACK[0], before restarting the PE. Clear the overflow status by writing to PMOVSCLR_EL0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter E6
Debug Reset and Powerdown Support

This chapter describes the reset and powerdown support in the Debug architecture. It contains the following sections:

- About Debug over powerdown.
- Power domains and debug.
- Core power domain power states.
- Emulating low-power states.
- Powerup request mechanism.
- Debug OS Save and Restore sequences.
- Reset and debug.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E6.1 About Debug over powerdown

Arm A-profile external debug defines a logical model for the hardware on which a PE executes. This hardware is logically split into the *Core power domain* and the *Debug power domain*, and the model contains descriptions of the states of those domains. See:

- Power domains and debug.
- Core power domain power states.

An implementation may allow power domains to be powered up and down independently. Debug over powerdown provides:

- A facility for software executing on the PE to save and restore the PE state on behalf of a self-hosted or external debugger or both. See Debug OS Save and Restore sequences.
- A facility for an external debugger to request power up of the Core power domain. See Powerup request mechanism.
- A facility for an external debugger, or software executing on the PE, to request emulation of powerdown of the Core power domain. See Emulating low-power states.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E6.2 Power domains and debug

Arm A-profile external debug has two logical power domains, each with its own reset:

- The Debug power domain contains the interface between the PE and the external debugger, and is powered up whenever an external debugger is connected to the SoC. It remains powered up while the external debugger is connected. When the Core power domain is completely off or in a low-power state, a debugger is permitted to access a register that is implemented in the Debug power domain. Registers in this domain are reset by an External Debug reset.
- The Core power domain contains the rest of the PE, and might be allowed to power up and power down independently of the Debug power domain.

Note

- The model of two logical power domains has an impact on the reset and access permission requirements of the debug programmers' model.
- The power domains are described as logical because the architecture defines the requirements but does not require two physical power domains. Any power domain split that meets the requirements of the programmers' model is a valid implementation.

The Core power domain contains several types of registers:

- Non-debug logic refers to all registers and logic that are not associated with debug.
- Self-hosted debug logic refers to registers and logic associated solely with the self-hosted debug aspects of the architecture.
- Shared debug logic refers to registers and logic associated with both the self-hosted and external debug aspects of the architecture.
- External debug logic refers to registers and logic associated solely with the external debug aspects of the architecture.

For information about which groups of registers and components are in each power domain, and which registers change power domain if FEAT_DoPD is implemented, see:

- Access permissions for the External debug interface registers.
- Cross-trigger interface registers.
- Management register access permissions.
- Access permissions for external views of the Performance Monitors.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E6.3 Core power domain power states

The Arm architecture does not define the power states of the PE as these are not normally visible to software. However, they are visible to the external debugger. Arm A-profile external debug uses a four logical power states model for the Core power domain. Mechanisms for entering a low-power state describes the architectural mechanisms for entering low-power states.

When the PE enters a low-power state other than Powerdown by executing a `WFI` or `WFE` instruction, it remains in that low-power state until it receives a wakeup event. See Mechanisms for entering a low-power state for the definition of wakeup events. When halting is allowed, an External Debug Request is a wakeup event. In addition, if the PE enters the Standby state for any reason, it will leave that state to service an External Debug Request debug event.

The four logical power states are as follows:

Normal The Core power domain is fully powered up and the debug registers are accessible.

Standby The Core power domain is on, but there are measures to reduce energy consumption. There can be other IMPLEMENTATION DEFINED measures the OS can take to enter standby.

The PE preserves the PE state, including the debug logic state. Changing from standby to normal operation does not involve a reset of the PE.

Standby is the least invasive OS energy saving state. Standby implies only that the PE is unavailable and does not clear any debug settings. For standby, the Debug architecture requires only the following:

- If the external debug interface is accessed, the PE must respond to that access. Arm recommends that, if the PE executed a `WFI` or `WFE` instruction to enter standby, then it does not retire that instruction.

Standby is transparent, meaning that to software and to an external debugger it is indistinguishable from normal operation.

Retention The PE state, including debug settings, is preserved in low-power structures, allowing the Core power domain to be at least partially turned off.

Changing from low-power retention to normal operation does not involve a reset of the PE. The saved PE state is restored on changing from low-power retention state to normal operation. If software has to use an IMPLEMENTATION DEFINED code sequence before entering, or after leaving, a retention state, this is referred to as a *software-visible retention state*. It is IMPLEMENTATION DEFINED whether the value of DBGPRCR.CORENPDRQ is set to its Cold reset value on leaving the software-visible retention state. See the description of DBGPRCR.CORENPDRQ for more information.

Note

- This model of retention does not include implementations where the PE exits the state in response to a debug register access. From the Debug architecture perspective, implementations like this are forms of standby.

Powerdown These measures must include the OS saving any PE state, including the debug settings, that must be preserved over powerdown.

If FEAT_DoubleLock is implemented, it is used during powerdown.

Changing from powerdown to normal operation must include:

- A Cold reset of the PE after the power level has been restored.
- The OS restoring the saved PE state.

External Debug Request debug events stay pending and debug registers in the Core power domain cannot be accessed.

An implementation might support enabling and disabling threads, either dynamically or once at reset time. Threads that are disabled in this way must appear to the external debugger as either:

- Powered off, meaning they are either:
  - In a powerdown state.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - In a retention state.

- Held in reset state.

Arm A-profile external debug uses a simpler two states model for the Debug power domain. The two states are:

Off The Debug power domain is turned off.

On The Debug power domain is turned on.

The available power states, including the cross-product of Core power domain and Debug power domain power states is IMPLEMENTATION DEFINED. Implementations are not required to implement all of these states and might include additional states. These additional states must appear to the debugger as one of the logical power states defined by this model. The control of power states is IMPLEMENTATION DEFINED.

Note

As a result, it is IMPLEMENTATION DEFINED whether it is possible for the Debug power domain to be on when the Core power domain is off.

If the Debug power domain is implemented but is not powered up when the Core power domain is powered up, the Reset Catch debug event and the OS Unlock Catch debug event are disabled.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E6.4 Powerup request mechanism

If a powerup request mechanism is implemented, asserting the powerup request requests the power controller to power up the Core power domain, and to emulate any subsequent powerdown requests, until the powerup request mechanism is deasserted.

E6.4.1 Powerup request mechanism if FEAT_DoPD is implemented

If FEAT_DoPD is implemented, the external debug component implements an OPTIONAL powerup request mechanism.

If the powerup request mechanism is implemented, Arm strongly recommends that the powerup request is a CoreSight Class `0x9` ROM table block that contains both:

- A parent entry for the debug registers of the PE.
- A parent entry for the PMU registers of the PE, if the OPTIONAL PMU with an external debug interface is implemented.

A parent entry of a component is an entry in a ROM table that either locates the component, or locates another ROM table that contains the parent entry for the component.

Note

The ROM table and any descendants might describe other debug components, including debug components for other PEs. The ROM table might have a parent entry in a second ROM table and that parent entry might also have a powerup request mechanism in the second ROM table. This applies recursively.

The parent entries for the debug components have the following properties:

For the debug registers and Performance Monitors registers: These components are in the Core power domain.

The POWERIDVALID bit is 1.

All parent entries must have the same IMPLEMENTATION DEFINED POWERID value.

Note

The IMPLEMENTATION DEFINED POWERID value does not need to be unique for each PE.

For the CTI registers: This component is in the Debug power domain.

The POWERIDVALID bit is IMPLEMENTATION DEFINED.

If the POWERIDVALID bit is 1, the entries must have a valid POWERID value.

Note

If the Core power domain can be powered down independently of the Debug power domain, Arm recommends the system implements an external debug component with a powerup request mechanism which can request the Core power domain to be powered up.

For more information about Coresight Class `0x9` ROM Tables, see *ARM® CoreSight™ Architecture Specification*.

On a Cold reset, if FEAT_DoPD is implemented, DBGPRCR_EL1, Debug Power Control Register.CORENPDRQ is set to an IMPLEMENTATION DEFINED choice between 0 and 1 if the powerup request is implemented and asserted, and 0 otherwise.

E6.4.2 Powerup request mechanism if FEAT_DoPD is not implemented

If FEAT_DoPD is not implemented, the bit EDPRCR.COREPURQ is the powerup request mechanism.

The control registers DBGPRCR_EL1, Debug Power Control Register.CORENPDRQ and EDPRCR.COREPURQ provide an interface between the power controller and the PE. They typically map directly to signals in the recommended external debug interface.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

On Cold reset, if FEAT_DoPD is not implemented, DBGPRCR_EL1, Debug Power Control Register.CORENPDRQ is set to the value of EDPRCR.COREPURQ.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E6.5 Emulating low-power states

DBGPRCR_EL1, Debug Power Control Register.CORENPDRQ and the powerup request mechanism can request the power controller to emulate states where the Core power domain is completely off or in a low-power state where the Core power domain registers cannot be accessed. This simplifies the requirements on software by sacrificing entirely realistic behavior.

If FEAT_DoPD is not implemented, EDPRSR.{SPD, PU} indicates the Core power domain power state. For more information, see:

- The DBGPRCR_EL1 and DBGPRCR System register descriptions.
- The EDPRCR and EDPRSR external debug register descriptions.
- Recommended External Debug Interface.

The measures to emulate powerdown are IMPLEMENTATION DEFINED. The ability of the debugger to access the state of the PE and the system might be limited as a result of the measures adopted.

In an emulated powerdown state, the debugger must be able to access all debug, PMU, CTI, and trace unit registers that are accessible on the external debug interface and are in one of:

- The Debug power domain.
- The Core power domain.
- When a trace unit with a separate trace unit Core power domain is implemented, and the trace unit Core power domain is powered on, the trace unit Core power domain.

That is, the debugger must be able to read and write to such registers without receiving errors. This allows an external debugger to debug the powerup sequence.

Arm recommends that any IMPLEMENTATION DEFINED registers that are on the external debug interface and in either the Core power domain or the Debug power domain are also accessible in an emulated powerdown state.

If FEAT_DoubleLock is implemented, `DoubleLockStatus() == FALSE` when DBGPRCR_EL1, Debug Power Control Register.CORENPDRQ == 1.

Otherwise, the behavior of the PE in emulated powerdown must be similar to that in a real powerdown state. In particular, the PE must not respond to other system stimuli, such as interrupts.

Example E6-1 and Example E6-2 are examples of two approaches to emulating powerdown.

Example E6-1 An example of emulating powerdown

The PE is held in Standby state, isolated from any system stimuli. It is IMPLEMENTATION DEFINED whether the PE can respond to debug stimuli such as an External Debug Request debug event.
If the PE can enter Debug state, then the external debugger is able to use the ITR to execute instructions, such as loads and stores. This causes the external debugger to interact with the system. If the external debugger restarts the PE, the PE leaves Standby state and restarts fetching instructions from memory.

Example E6-2 Another example of emulating powerdown

The PE is held in Warm reset. This limits the ability of an external debugger to access the resources of the PE. For example, the PE cannot be put into Debug state.

On exit from emulated powerdown the PE is reset. However, the debug registers that are only reset by a Cold reset must not be reset. Typically this means that a Warm reset is substituted for the Cold reset. As such, the effect of accessing any register that is reset by a Warm reset while the PE is in the emulated powerdown state will have an IMPLEMENTATION DEFINED effect on that register.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

- Warm reset and Cold reset have different effects apart from resetting the debug registers. In particular, RMR_ELx is reset by a Cold reset and controls the reset state on a Warm reset. This means that if a Cold reset is substituted by a Warm reset, the behavior of the reset code might be different.
- The timing effects of powering down are typically not factored in the powerdown emulation. Examples of these timing effects are clock and voltage stabilization.
- Emulation does not model the state lost during powerdown, meaning that it might mask errors in the state storage and recovery routines.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E6.6 Debug OS Save and Restore sequences

From the introduction of Armv8-R, the following registers provide the OS Save and Restore mechanism:

- The *OS Lock Access Register*, OSLSR_EL1, OS Lock Status Register, locks the OS Lock to restrict access to debug registers before starting an OS Save sequence, and unlocks the OS Lock after an OS Restore sequence.
- The *OS Lock Status Register*, OSLSR_EL1, OS Lock Status Register, shows the status of the OS Lock.
- The PE can be configured to generate an OS Unlock Catch debug event when the OS Lock is unlocked.
- If FEAT_DoubleLock is implemented, the OS Double Lock locks out an external debug interface entirely. This is only used immediately before a powerdown sequence.

See also:

- FEAT_DoubleLock
- Reset and debug
- Example OS Save and Restore Sequences

E6.6.1 EDPRSR.{DLK, SPD, PU} and the Core power domain

If FEAT_DoPD is not implemented, a debugger uses EDPRSR.{DLK, SPD, PU} to determine whether registers in the Core power domain can be accessed, and whether their state has been lost since the last time the register was read.

Table E6-1 Interpretation of the EDPRSR.{DLK, SPD, PU} bits

EDPRSR			Core power domain			Notes
DLK	SPD	PU	Power	Accesses	State lost	
0	0	1	On	OK	No	-
0	1	1	On	OK	Yes	SPD is cleared to 0 following the read.
1	X	1	On	Error	Not known	FEAT_DoubleLock is implemented and `DoubleLockStatus()` == TRUE. Software locks the OS Double Lock before removing power.
X	1	0	Off	Error	Yes	A Cold reset will be asserted on exiting powerdown state, but not on exiting low-power retention state.
X	0	0	Not known	Error	Not known	

If FEAT_DoPD is implemented, accesses to EDPRSR return an error when the Core power domain is off or in a retention state, meaning successful reads of EDPRSR always return 1 for EDPRSR.PU.

When FEAT_Debugv8p4 is implemented, and whenever FEAT_DoubleLock is not implemented, EDPRSR.DLK is always 0.

If FEAT_DoubleLock is not implemented, `DoubleLockStatus()` always returns FALSE.

If the Core power domain is powered up and `DoubleLockStatus()` == TRUE, then:

- When FEAT_Debugv8p2 is not implemented, EDPRSR.{DLK, SPD, PU} can read either {1, UNKNOWN, 1} or {UNKNOWN, 0, 0}.
- When FEAT_Debugv8p2 is implemented, and FEAT_Debugv8p4 is not implemented, EDPRSR.{DLK, SPD, PU} can only read {UNKNOWN, 0, 0}.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E6.6.2 EDPRSR.SPD when the Core domain is in either retention or powerdown state

If FEAT_DoPD is not implemented, when the Core power domain is in either the retention or powerdown state, EDPRSR.SPD is not cleared following a read of EDPRSR and it is IMPLEMENTATION DEFINED whether:

- EDPRSR.SPD shows whether the state of the debug registers in the Core power domain has been lost since the last time that EDPRSR was read. This means that:
  - When the Core power domain is in the powerdown state, EDPRSR.SPD is RAO, this indicates that the state of the debug registers has been lost.
  - When the Core power domain is in the retention state, EDPRSR.SPD indicates whether the state of the debug registers was lost before the Core power domain entered retention state.
- EDPRSR.SPD is RAZ, and:
  - On leaving the powerdown state, EDPRSR.SPD is set to 1 which indicates that the state of the debug registers has been lost.
  - On leaving the retention state, EDPRSR.SPD reverts the value it had on entering the retention state.

Note

If FEAT_DoPD is implemented, accesses to EDPRSR return an error when the Core power domain is off or in a retention state.

E6.6.3 EDPRSR.{DLK, R} and reset state

If FEAT_DoPD is implemented, accesses to EDPRSR return an error when the Core power domain is off or in a retention state, meaning successful reads of EDPRSR always return 1 for EDPRSR.PU.

When FEAT_Debugv8p4 is implemented, and whenever FEAT_DoubleLock is not implemented, EDPRSR.DLK is always 0.

If FEAT_DoubleLock is not implemented, `DoubleLockStatus()` always returns FALSE.

If FEAT_DoubleLock is implemented and enabled, the behavior of all registers and fields except EDPRSR.DLK is the same as their behavior if FEAT_Debugv8p4 is not implemented.

If FEAT_Debugv8p4 is implemented EDPRSR.DLK is always 0 and does not give any information about the OS Double Lock.

EDPRSR.R is UNKNOWN when `DoubleLockStatus()` == TRUE. OSDLR_EL1.DLK is cleared to 0 by a reset. If the Core power domain is powered up and entered reset state with the OS Double Lock locked, it is CONSTRAINED UNPREDICTABLE whether a read of EDPRSR while the PE is in reset state returns:

- EDPRSR.{DLK, R, PU} == {1, UNKNOWN, 1} indicating that the OS Double Lock is locked. This is not permitted from Armv8.2.
- EDPRSR.{DLK, R, PU} == {0, 1, 1} indicating that the PE is in reset state.
- EDPRSR.{DLK, R, PU} == {UNKNOWN, UNKNOWN, 0} indicating that the registers in the Core power domain cannot be accessed because the OS Double Lock is locked.

If the PE was powered up and the OS Double Lock was unlocked when the PE was reset, then EDPRSR.{DLK, R, PU} reads as {0, 1, 1} while the PE is in reset state.

On leaving reset state, EDPRSR.{DLK, R} reads as {0, 0}.

E6.6.4 Debug registers to save over powerdown

Table E6-2 shows the different requirements for self-hosted debug over powerdown and external debug over powerdown:

- The column labeled Self-hosted lists registers that software must preserve over powerdown so that it can support self-hosted debug over powerdown. This does not require use of the OS Save and Restore mechanism.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The column labeled External lists registers that software must preserve over powerdown so that it can support external debug over powerdown. This requires use of the OS Save and Restore mechanism:
  - Some external debug registers are not normally accessible to software executing on the PE. Additional debug registers are provided that give software the required access to save and restore these external debug registers when OSLSR_EL1, OS Lock Status Register.OSLK is locked. These registers include OSECCR_EL1, OS Lock Exception Catch Control Register, OSDTRRX_EL1, OS Lock Data Transfer Register, Receive, and OSDTRTX_EL1, OS Lock Data Transfer Register, Transmit.
- Some registers might only present in some implementations, or might not be accessible at all Exception levels or in Non-secure state. .

Table E6-2 does not include registers for the OPTIONAL Trace and Performance Monitor extensions.

Table E6-2 Debug registers to save over powerdown

Register in AArch64 state	Self-hosted	External
MDSCR_EL1	Yes	Yes
DBGBVR<n>_EL1	Yes	Yes
DBGBCR<n>_EL1	Yes	Yes
DBGWVR<n>_EL1	Yes	Yes
DBGWCR<n>_EL1	Yes	Yes
MDCR_EL2	Yes	-
DBGCLAIMSET_EL1 DBGCLAIMCLR_EL1	-	Yes
OSECCR_EL1	-	Yes
OSDTRRX_EL1 OSDTRTX_EL1	-	Yes

E6.6.5 OS Save sequence

To preserve the debug logic state over a powerdown, the state must be saved to nonvolatile storage. This means the OS Save sequence must:

1. Lock the OS Lock by writing 1 to OSLAR_EL1.OSLK.
2. Execute an `ISB` instruction.
3. Walk through the debug registers listed in Debug registers to save over powerdown and save the values to the nonvolatile storage.

If the FEAT_DoubleLock is implemented, before removing power from the Core power domain, software must:

1. Lock the OS Double Lock by writing 1 to OSDLR_EL1.DLK.
   If FEAT_DoubleLock is not implemented, OSDLR_EL1 ignores writes.
2. Execute a Context Synchronization event.

E6.6.6 OS Restore sequence

After a powerdown, the OS Restore sequence must perform the following steps to restore the debug logic state from the non-volatile storage:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

1. Lock the OS Lock, as described in OS Save sequence. The OS Lock is generally locked by the Cold reset, but this step ensures that it is locked.
2. Execute an `ISB` instruction.
3. Write 0 to MDSCR_EL1 to ensure that, if an external debugger clears the OS Lock before the end of this sequence, no debug exceptions are generated.
4. Walk through the debug registers listed in Debug registers to save over powerdown, and restore the values from the nonvolatile storage. The last register to be restored must be MDSCR_EL1.
5. Execute an `ISB` instruction.
6. Unlock the OS Lock by writing 0 to OSLAR_EL1.OSLK.
7. Execute a Context Synchronization event.

Note

The OS Restore sequence overwrites the debug registers with the values that were saved. If there are valid values in these registers immediately before the restore sequence, then those values are lost.

E6.6.7 Debug behavior when the OS Lock is locked

The main purpose of the OS Lock is to prevent updates to debug registers during an OS Save or OS Restore operation. The OS Lock is locked on a Cold reset.

When the OS Lock is locked:

- Access to debug registers through the System register interface is mainly unchanged except that:
  - Certain registers are read and written without side-effects.
  - Fields in MDSCR_EL1, Monitor Debug System Control Register and OSECCR_EL1, OS Lock Exception Catch Control Register that are normally read-only become read/write.

  This allows the state to be saved or restored. For more information, see the relevant register description in External Debug Register Descriptions.
- Access to debug registers by the external debug interface is restricted to prevent an external debugger modifying the registers that are being saved or restored. For more information, see External debug interface register access permissions summary.
- Debug exceptions, other than Breakpoint Instruction exceptions are not generated.
- Breakpoint and Watchpoint debug events are not generated. The OS Lock has no effect on Breakpoint Instruction exceptions and other debug events.

E6.6.8 Debug behavior when the OS Lock is unlocked

When the OS Lock is unlocked, the PE sets EDESR.OSUC to 1 if the OS Unlock Catch debug event is enabled and the PE is in Non-debug state, meaning the OS Unlock Catch debug event becomes pending. See OS Unlock Catch debug event.

E6.6.9 Debug behavior when the OS Double Lock is locked

If the FEAT_DoubleLock is implemented, software locks the OS Double Lock immediately before a powerdown sequence.

The OS Double Lock ensures that it is safe to remove core power by forcing the debug interfaces to be quiescent.

When `DoubleLockStatus()` == TRUE:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The external debug interface has only restricted access to the debug registers, so that it is quiescent before removing power. See External debug interface register access permissions summary.
- Debug exceptions, other than Breakpoint Instruction exceptions, are not generated.
- Halting is prohibited. See Halting allowed and halting prohibited.

Note

Pending Halting debug events might be lost when core power is removed.

- No asynchronous debug events are `WF*` wakeup events.

If the FEAT_DoubleLock is not implemented, the PE ensures these conditions are met before allowing power to be removed.

Software must synchronize the update to OSDLR_EL1, OS Double Lock Register before it indicates to the system that core power can be removed. The interface between the PE and its power controller is IMPLEMENTATION DEFINED.

Typically software indicates that core power can be removed by entering the Wait For Interrupt state. This means that software must explicitly synchronize the OSDLR_EL1, OS Double Lock Register update before issuing the `WFI` instruction.

OSDLR_EL1, OS Double Lock Register.DLK is ignored and `DoubleLockStatus()` == FALSE if either:

- The PE is in Debug state.
- DBGPRCR_EL1, Debug Power Control Register.CORENPDRQ is set to 1.

Note

It is possible to enter Debug state with OSDLR_EL1, OS Double Lock Register.DLK set to 1. This is because a Context Synchronization event is required to ensure the OS Double Lock is locked, meaning that Debug state might be entered before the OSDLR_EL1, OS Double Lock Register update is synchronized.

Because OSDLR_EL1, OS Double Lock Register.DLK is ignored when DBGPRCR_EL1, Debug Power Control Register.CORENPDRQ is set to 1, an external debugger can write to DBGPRCR_EL1, Debug Power Control Register.CORENPDRQ, and the FEAT_DoubleLock is not always implemented, software must not rely on using the OS Double Lock to disable debug exceptions or to prohibit halting, or both. Arm deprecates use of the OS Double Lock for these purposes, and instead recommends that software:

- Uses the OS Lock to disable debug exceptions during save or restore sequences.
- Uses the debug authentication interface to prohibit halting and external debug access to debug registers at times other than immediately prior to removing power.

As the purpose of the OS Double Lock is to ensure that it is safe to remove core power, if the FEAT_DoubleLock is implemented, it is important to avoid race conditions that defeat this purpose. Arm recommends that:

- Once the write to OSDLR_EL1, OS Double Lock Register.DLK has been synchronized by a Context Synchronization event and `DoubleLockStatus()` == TRUE, a PE must:
  - Not allow a debug event generated before the Context Synchronization event to cause an entry to Debug state or act as a wakeup event for a `WFI` or `WFE` instruction after the Context Synchronization event has completed.
  - Complete any external debug access started before the Context Synchronization event by the time the Context Synchronization event completes.

    Note

    A debug register access might be in progress when software sets OSDLR_EL1, OS Double Lock Register.DLK to 1. An implementation must not permit the synchronization of locking the OS Double Lock to stall indefinitely while waiting for that access to complete. This means that any debug register access that is in progress when software sets OSDLR_EL1, OS Double Lock Register.DLK to 1 must complete or return an error in finite time.

- If a write to DBGPRCR_EL1, Debug Power Control Register or EDPRCR made when OSDLR_EL1, OS Double Lock Register.DLK == 1 changes DBGPRCR_EL1, Debug Power Control Register.CORENPDRQ or

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EDPRCR.CORENPDRQ from 1 to 0, meaning `DoubleLockStatus()` changes from FALSE to TRUE, then before signaling to the system that the CORENPDRQ field has been cleared and emulation of powerdown is no longer requested, meaning the system can remove core power, the PE must ensure that all the requirements for `DoubleLockStatus()` == TRUE listed in this section are met.

In a standard OS Save sequence, the OS Lock is locked before the OS Double Lock is locked. This means that writes to CORENPDRQ are ignored by the time the OS Double Lock is locked. However, if `DoubleLockStatus()` == FALSE, an external debugger can clear the OS Lock at any time, and then write to EDPRCR.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E6.7 Reset and debug

All registers in the Core power domain are either:

- Reset by both a Cold and a Warm reset.
- Reset only by a Cold reset and are not changed by a Warm reset.

For more information, see Resets and power domains.

All registers in the Debug power domain are reset by an External Debug reset.

If FEAT_SPMU is implemented, the reset signals for a System PMU are IMPLEMENTATION DEFINED.

Figure E6-1 shows this reset scheme. The following three reset signals are an example implementation of the reset scheme:

- **CORERESET**, which must be asserted for a Warm reset.
- **CPUPORESET**, which must be asserted for a Cold reset.
- **PRESETDBG**, which must be asserted for an External Debug reset.

As shown in the figure, the external debug logic is split between the Debug power domain and the Core power domain.

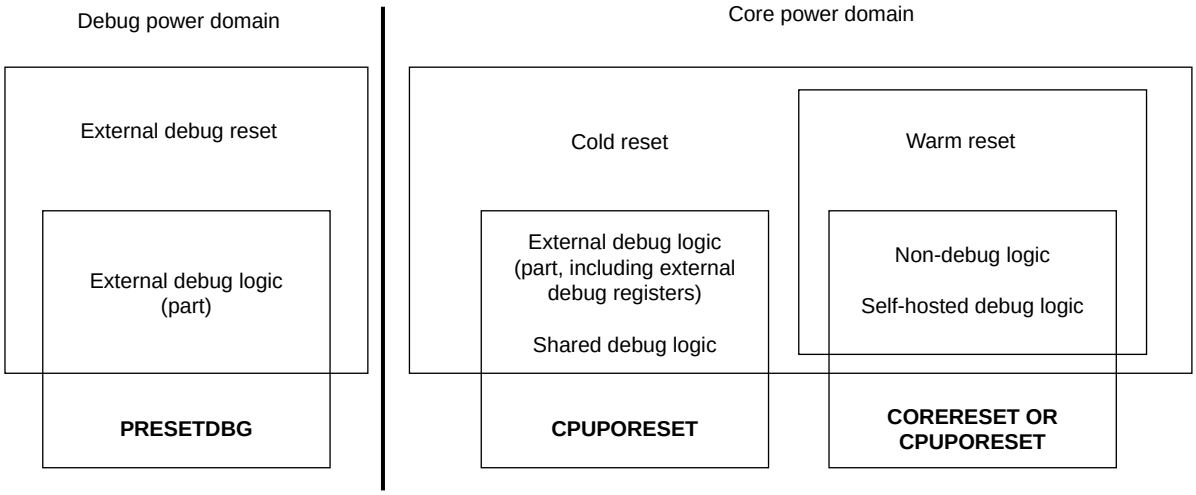


Figure E6-1 Power and reset domains

For more information about power domains and power states, see Power domains and debug.

When power is first applied to the Debug power domain, **PRESETDBG** must be asserted.

When power is first applied to the Core power domain, **CPUPORESET** must be asserted.

Note

In this scheme, logic in the Warm reset domain is reset by asserting either **CORERESET** or **CPUPORESET**. This implies a particular implementation style that permits these approaches.

CPUPORESET is not normally asserted on moving from a low-power state, where power has not been removed, to a full-power state. This can occur, for example, on exiting a low-power retention state. See also Emulating low-power states and the EDPRSR register description.

E6.7.1 External debug interface accesses to registers in reset

If a reset signal is asserted and the external debug interface:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Writes a register, or indirectly writes a register or register field as a side-effect of an access:
  - Then, if the register or register field is reset by that reset signal, it is CONSTRAINED UNPREDICTABLE whether the register or register field takes the reset value or the value written. The reset value might be UNKNOWN.
  - Otherwise, the register or register field takes the value that is written.
- Reads a register, or indirectly reads a register or register field, as part of an access:
  - Then, if the register or register field is reset by that reset signal, the value returned in UNKNOWN.
  - Otherwise, the value of the register or register field is returned.

It is IMPLEMENTATION DEFINED whether any register can be accessed when External Debug reset is being asserted. The result of these accesses is IMPLEMENTATION DEFINED.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter E7
The PC Sample-based Profiling Extension

This chapter describes the OPTIONAL PC Sample-based Profiling Extension that provides a non-invasive external debug component.

It contains the following section:

- About the PC Sample-based Profiling Extension.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E7.1 About the PC Sample-based Profiling Extension

The PC Sample-based Profiling Extension is an OPTIONAL extension that provides coarse-grained, non-invasive profiling by an external debugger. See also Non-invasive behavior.

When the PC Sample-based Profiling Extension is implemented, the PE implements a PC sample register and one or more context sample registers as external registers. Reads of the PC sample register return a PC sample value and capture the PE context in the context sample registers. For example, tools can use these samples to populate a statistical model of the performance of software executing on the PE.

Note

Data returned by periodic sampling of PC Sample-based Profiling registers is sufficient to allow tools to estimate the distribution of time spent executing software on the PE.
The delay between an instruction being executed by the PE and its address appearing in the PC Sample Register is not defined, and the architecture does not require that the sampled instruction was recently executed. For example, if a piece of software executes a load instruction that reads the PC Sample Register of the PE it is running on, there is no guaranteed relationship between the address of the load instruction and the value read. The PC Sample Register is intended only for use by an external agent to provide statistical information for software profiling.

It must be possible to sample references to branch targets. It is IMPLEMENTATION DEFINED whether references to other instructions can be sampled. The branch target for a conditional branch instruction that fails its condition check is the instruction that follows the conditional branch instruction. The branch target for an exception is the exception vector address.

Note

In the Armv7 PC Sample-based Profiling Extension, an offset was applied to the sampled Program Counter value and this offset and the instruction set state indicated in bits [1:0] of the sampled value. From the introduction of the Armv8 PC Sample-based Profiling Extension, the sampled value is the address of an instruction that has executed, with no offset and no indication of the instruction set state.

- Controlling the PC Sample-based Profiling Extension.
- Registers implemented by the PC Sample-based Profiling Extension.
- Permitted behavior that might make the PC Sample-based profiling registers UNKNOWN.
- Pseudocode description of PC Sample-based Profiling.

E7.1.1 Controlling the PC Sample-based Profiling Extension

PC Sample-based Profiling is controlled by the IMPLEMENTATION DEFINED authentication interface `ExternalNoninvasiveDebugEnabled()`.

PC Sample-based Profiling is allowed if both of the following apply:

- `ExternalNoninvasiveDebugEnabled()` == TRUE.
- `ExternalSecureNoninvasiveDebugEnabled()` == TRUE.

The state of the IMPLEMENTATION DEFINED authentication interface is visible through DBGAUTHSTATUS_EL1. See Recommended authentication interface.

E7.1.1.1 Suspending and activating PC Sample-based Profiling

The state of PC Sample-based Profiling at Warm reset is an IMPLEMENTATION DEFINED one of suspended or active.

When all of the following apply, reading PMPCSR[31:0] or PMPCSR[63:0] makes PC Sample-based Profiling active:

- PC Sample-based Profiling is suspended.
- PC sampling is allowed.

A subsequent external read of PMPCSR will observe the updated state.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

There is no mechanism to return to suspended state, other than Warm reset.

E7.1.2 Registers implemented by the PC Sample-based Profiling Extension

If FEAT_PCSRv8p2 is implemented and the 64-bit external PMU programmers' model extension is not implemented, the following PC Sample-based Profiling Extension registers can be implemented in the Performance Monitors memory-mapped register space:

- PMCID1SR and PMCID2SR.
- PMPCSR.
- PMVIDSR.

When the 64-bit external PMU programmers' model extension is implemented, the PC Sample-based Profiling Extension registers are:

- PMCCIDSR.
- PMPCSR.
- PMVCIDSR.

See Performance Monitors external register views.

If FEAT_PCSRv8p2 is implemented but the Performance Monitors Extension is not implemented, then the PC Sample-based Profiling Extension registers are implemented in their own memory-mapped register space, within the area that is reserved for the Performance Monitors, see Table E7-1. If CoreSight compliance is required:

- The management registers are defined as in The Arm® Architecture Reference Manual, for A-profile architecture.
- The support for PC Sample-based profiling is defined in the following registers:
  - PMDEVTYPE.MAJOR has the value `0x0`.
  - PMDEVARCH.ARCHID has the value `0x0A10` or `0x0A20`.

For more information, see PMDEVARCH.

Table E7-1 PC Sample-based Profiling register map without the Performance Monitors Extension

Offset	Description
`0x200`	PMPCSR[31:0]
`0x204`	PMPCSR[63:32]
`0x208`	PMCID1SR
`0x20C`	PMVIDSR
`0x220`	PMPCSR[31:0] (alias)
`0x224`	PMPCSR[63:32] (alias)
`0x228`	PMCID1SR (alias)
`0x22C`	PMCID2SR
`0x600-0x6FC`	IMPLEMENTATION DEFINED
`0xE80-0xEFC`	IMPLEMENTATION DEFINED for CoreSight compliance
`0xFF0-0xFFc`	Management and CoreSight compliance registers

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E7.1.3 Permitted behavior that might make the PC Sample-based profiling registers UNKNOWN

The architecture permits IMPLEMENTATION DEFINED extensions to external debug to define mechanisms that make the values of the PC Sample-based profiling registers UNKNOWN. However, it requires that any such mechanism is disabled by default. This means that powerup or a hard reset of the PE must leave the PE in a state where the PC Sample-based Profiling Extension, if implemented, exhibits its architecturally-defined behavior.

Note

A mechanism that, when enabled, makes the PC Sample-based profiling registers UNKNOWN might use other sample-based profiling events that are appropriate for a use that is independent of PC Sample-based Profiling.

If no branch instruction has been retired since the PE left Debug state, suspended state, or a state where PC Sample-based profiling is prohibited, then the sampled value is UNKNOWN.

If all of the following apply, then a read of PMPCSR[31:0] or PMPCSR[63:0] is permitted but not required to return `0xFFFF_FFFF` for PMPCSR[31:0].

- The PE has not retired any branch instruction since leaving a state where PC Sample-based Profiling was suspended.
- Sampling on reads is not disabled.

This includes a read of PMPCSR[31:0] or PMPCSR[63:0] that indirectly makes PC Sample-based Profiling active.

E7.1.4 Pseudocode description of PC Sample-based Profiling

When FEAT_PCSRv8p2 is implemented, the functionality is described by the pseudocode functions:

- `CreatePCSample()`, which populates a variable of type `PCSample`.
- `Read_PMPCSR[]`, which writes a PC Sample to the PMPCSR and associated registers.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter E8
About the External Debug Registers

This chapter provides some additional information about the external debug registers. It contains the following sections:

- Relationship between external debug and System registers.
- Endianness and supported access sizes.
- Synchronization of changes to the external debug registers.
- Memory-mapped accesses to the external debug interface.
- External debug interface register access permissions.
- External debug interface registers.
- Cross-trigger interface registers.
- External debug register resets.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E8.1 Relationship between external debug and System registers

Table E8-1 shows the relationship between external debug registers and System registers. Where no relationship exists, the registers are not listed.

Table E8-1 Equivalence between external debug and System registers

External debug register	System register	Notes
DBGDTRRX_EL0	DBGDTRRX_EL0	See also Summary of System register accesses to the DCC
DBGDTRTX_EL0	DBGDTRTX_EL0	
OSLAR_EL1	OSLAR_EL1	-
DBGBVR<n>_EL1[31:0] DBGBVR<n>_EL1[63:32]	DBGBVR<n>_EL1[31:0] DBGBVR<n>_EL1[63:32]	-
DBGBCR<n>_EL1[31:0] DBGBCR<n>_EL1[63:32]	DBGBCR<n>_EL1[31:0] DBGBCR<n>_EL1[63:32]	-
DBGWVR<n>_EL1[31:0] DBGWVR<n>_EL1[63:32]	DBGWVR<n>_EL1[31:0] DBGWVR<n>_EL1[63:32]	-
DBGWCR<n>_EL1[31:0] DBGWCR<n>_EL1[63:32]	DBGWCR<n>_EL1[31:0] DBGWCR<n>_EL1[63:32]	-
DBGCLAIMSET_EL1	DBGCLAIMSET_EL1	-
DBGCLAIMCLR_EL1	DBGCLAIMCLR_EL1	-
DBGAUTHSTATUS_EL1	DBGAUTHSTATUS_EL1	Read-only
EDSCR	MDSCR_EL1	
EDECCR	OSECCR_EL1	Applies when the OS Lock is locked
MIDR_EL1	MIDR_EL1	Read-only copies of Processor ID Registers
EDDEVAFF0 EDDEVAFF1	MPIDR_EL1[31:0][a] MPIDR_EL1[63:32][a]	Read-only copies of system ID registers

a. This is a word of a 64-bit register.

In addition:

- EDSCR.{TXfull, RXfull} are read-only aliases for MDCCSR_EL0, Monitor DCC Status Register.{TXfull, RXfull}.
- EDPRCR.CORENPDRQ is a read/write alias for DBGPRCR_EL1, Debug Power Control Register.CORENPDRQ.
- EDPRSR.OSLK is a read-only alias for OSLSR_EL1, OS Lock Status Register.OSLK.
- If the FEAT_DoubleLock is implemented, EDPRSR.DLK is a read-only function of OSDLR_EL1, OS Double Lock Register.DLK.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E8.2 Endianness and supported access sizes

The debug registers, Performance Monitors registers, and CTI registers are implemented as memory-mapped peripherals. The Arm architecture requires memory-mapped peripherals to be little-endian.

The memory access sizes supported by any peripheral is IMPLEMENTATION DEFINED by the peripheral. For accesses to the debug registers, Performance Monitors registers, and CTI registers, implementations must:

- Comply with the requirements of Supported access sizes.
- Support word-aligned 32-bit accesses to access 32-bit registers or either half of a 64-bit register mapped to a doubleword-aligned pair of adjacent 32-bit locations, even if all components with direct memory access to the memory-mapped peripheral support making 64-bit accesses.

Note

These requirements mean that a system implementing the debug registers using a 32-bit bus, such as an AMBA APB3, with a wider system interconnect must implement a bridge between the system and the debug bus that can split 64-bit accesses.

For accesses from the external debug interface, the size of an access is determined by the interface. For an access from an ADIv5-compliant Memory Access Port, MEM-AP, this is specified by the MEM-AP CSW register.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E8.3 Synchronization of changes to the external debug registers

This section describes the synchronization requirements for the external debug interface.

For more information on how these requirements affect debug, see:

- Synchronization and debug exceptions for exceptions taken from AArch64 state.
- Synchronization and Halting debug events.
- Synchronization of DCC and ITR accesses.

This section refers to accesses from the external debug interface as external reads and external writes. It refers to accesses to System registers as direct reads, direct writes, indirect reads, and indirect writes.

Note

Synchronization requirements for AArch64 System registers define direct read, direct write, indirect read, and indirect write, and classifies external reads as indirect reads, and external writes as indirect writes.

For general information about synchronization, access completion, ordering, and observability, see Synchronization of memory-mapped registers.

Writes to the same register are serialized, meaning they are observed in the same order by all observers, although some observers might not observe all of the writes. With the exception of DBGBCR<n>_EL1, DBGBVR<n>_EL1, DBGWCR<n>_EL1, and DBGWVR<n>_EL1, external writes to different registers are not necessarily observed in the same order by all observers as the order in which they complete.

Synchronization of DCC and ITR accesses describes the synchronization requirements for the DCC and ITR.

Changes to the IMPLEMENTATION DEFINED authentication interface are external writes to the authentication status registers by the Requester of the authentication interface. See Synchronization and the authentication interface.

The external agent must be able to guarantee completion of a write. For example by:

- Marking the memory as Device-nGnRnE and executing a `DSB` barrier, if the system supports this property.
- Reading back the value written.
- Some guaranteed property of the connection between the PE and the external agent.

Note

For an external Debug Access Port, access completion is an IMPLEMENTATION DEFINED property. For a CoreSight system using APB-AP to access a debug APB, accesses complete in order.

However, the external agent cannot force synchronization of completed writes without halting the PE. Executing an `ISB` instruction, either in Debug state or in Non-debug state, and exiting from Debug state forces synchronization. If the PE is in Debug state, executing an `ISB` instruction is guaranteed to explicitly synchronize any external reads, external writes, and changes to the authentication interface that are ordered before the external write to EDITR.

For any given observer, external writes to the following register groups are guaranteed to be observable in the same order in which they complete:

- The breakpoint registers, DBGBCR<n>_EL1 and DBGBVR<n>_EL1.
- The watchpoint registers, DBGWCR<n>_EL1 and DBGWVR<n>_EL1.

This guarantee applies only to external writes to registers within one of these groups. There is no guarantee regarding the ordering of the observability of external writes within these groups with respect to external writes to registers, for example EDSCR, or between breakpoints and watchpoints, including watchpoints linked to context matching breakpoints.

Note

This means that a debugger can rely on the external writes to be observed in the same order in which they complete. It does not mean that a debugger can rely on the external writes being observed in finite time.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

In a simple sequential execution an indirect write that occurs as a side-effect of an access happens atomically with the access, meaning no other accesses are allowed between the register access and its side-effect.

If two or more interfaces simultaneously access a register, the behavior must be as if the accesses occurred atomically and in any order. This is described in Examples of the synchronization of changes to the external debug registers.

Note

Where simultaneous writes occur, once an order has been established by the implementation, some registers have behavior that is incompatible with the implementation simply dropping the older write and only performing the actions of the younger write. In these cases, the implementation must ensure that the register update is consistent with both writes having occurred. For example, writing one to write-one-to-clear or write-one-to-set fields in the older write can produce effects that persist even after writing zero to the same field in the younger write. Examples of registers with this behavior include DBGCLAIMSET_EL1, DBGCLAIMCLR_EL1, PMCR_EL0, PMOVSSET_EL0, PMOVSCLR_EL0, PMCNTENSET_EL0, PMCNTENCLR_EL0, PMINTENSET_EL1, PMINTENCLR_EL1, PMSWINC_EL0, and PMZR_EL0, all of which are simultaneously accessible via external debug and System register access.

E8.3.1 Synchronization and the authentication interface

Changes to the authentication interface are indirect writes to the state of the PE by the Requester of the authentication interface.

For an external debug interface read of any Authentication Status register, or an indirect read of the authentication interface made in determining the response to a subsequent external debug interface access, a change on the authentication interface must be observable following a subsequent explicit Context Synchronization event, and:

- It is IMPLEMENTATION DEFINED whether a change is guaranteed to be observable in finite time.
- It is IMPLEMENTATION DEFINED whether a change is guaranteed to be observable following an entry to Debug state.

For a System register read of DBGAUTHSTATUS_EL1, a change on the authentication interface is guaranteed to be observable only after a Context Synchronization event.

Note

- In some systems, the authentication interface is fixed by configuration or is changed under the control of software. These systems can require explicit synchronization for any change to the authentication interface.
- In other systems, the authentication interface is controlled dynamically by an external agent. In these systems, it is desirable that changes to the authentication interface do not require explicit synchronization by software executing on the PE to be observable by subsequent external debug interface accesses, and are either observable in finite time or are synchronized by entry to Debug state. Otherwise, there are scenarios where a debugger is not able to halt and debug the system.

E8.3.2 Examples of the synchronization of changes to the external debug registers

Example E8-1, Example E8-2, and Example E8-3 show the synchronization of changes to the external debug registers.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example E8-1 Order of synchronization of Breakpoint and Watchpoint register writes

Initially DBGBVR<n>_EL1 is `0x8000` and DBGBCR<n>_EL1 is `0x0181`. This means that a breakpoint is enabled on the halfword T32 instruction at address `0x8000`.
A sequence of external writes occurs in the following order:

1. `0x0000` is written to DBGBCR<n>_EL1, disabling the breakpoint.
2. `0x9000` is written to DBGBVR<n>_EL1[31:0].
3. `0x0061` is written to DBGBCR<n>_EL1, enabling a breakpoint on the halfword at address `0x9002`.

The external writes must be observable to indirect reads in the same order as the external writes complete. This means that at no point is there a breakpoint enabled on either of the halfwords at address `0x8002` and `0x9000`. Similarly a breakpoint or watchpoint must be disabled:

- If both halves of a 64-bit address have to be updated.
- If any of the DBGBCR<n>_EL1 or DBGWCR<n>_EL1 fields are modified at the same time as updating the address.

Example E8-2 Simultaneous accesses to DTR registers

Initially EDSCR.{TXfull, TXU, ERR} are 0. Then:

- `0x0DCCDA7A` is directly written to DBGDTRTX_EL0 by an `MSR` instruction.
- DBGDTRTX_EL0 is indirectly read by the external debug interface.

These accesses might happen at the same time and in any order.
If the direct write of `0x0DCCDA7A` to DBGDTRTX_EL0 is handled first, then:

- The external debug interface read of DBGDTRTX_EL0 clears EDSCR.TXfull to 0.
- EDSCR.{TXU, ERR} are unchanged.
- The external debug interface read returns `0x0DCCDA7A`.

If the indirect read of DBGDTRTX_EL0 by the external debug interface is handled first, then:

- The external debug interface read of DBGDTRTX_EL0 causes an underrun and as a result EDSCR.{TXU, ERR} are both set to 1.
- The external debug interface returns an UNKNOWN value.
- Writing `0x0DCCDA7A` to DBGDTRTX_EL0 sets DTRTX to `0x0DCCDA7A` and EDSCR.TXfull to 1.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example E8-3 Simultaneous writes to CLAIM registers

Initially all CLAIM tag bits are 0. Then:

- `0x01` is written to DBGCLAIMSET_EL1 by a direct write, followed by an explicit Context Synchronization event.
- `0x02` is written to DBGCLAIMSET_EL1 by an external write.

These events might happen at the same time and in either order.
After this:

- DBGCLAIMCLR_EL1 is read by a direct read.
- DBGCLAIMCLR_EL1 is read by an external read.

In this case, a direct read can return either `0x01` or `0x03`, and the external read can return either `0x02` or `0x03`.
The only permitted final result for the CLAIM tags is the value `0x03`, because this would be the result regardless of whether `0x01` or `0x02` is written first. This is because the external write is guaranteed to be observable to a direct read in finite time. See Synchronization requirements for AArch64 System registers.
It is not possible for a direct read to return `0x01` and the external read to return `0x02`, because the writes to DBGCLAIMCLR_EL1 are serialized.
In the following scenario, there is only one permitted result. Both observers observe the value `0x03`, and then, at the same time, two writes occur:

- `0x04` is written to DBGCLAIMSET_EL1 by a direct write, followed by an explicit Context Synchronization event.
- `0x01` is written to DBGCLAIMCLR_EL1 by an external write.

In this case, the only permitted final result for the CLAIM tags is the value `0x06`.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E8.4 Memory-mapped accesses to the external debug interface

Support for memory-mapped access to the external debug interface is OPTIONAL. When memory-mapped access to the external debug interface is supported, the external debug interface is accessed as a little-endian memory-mapped peripheral.

If the external debug interface is CoreSight compliant, then an OPTIONAL Software Lock can be implemented for memory-mapped accesses to each component.

The Software Lock is OPTIONAL and deprecated. If FEAT_Debugv8p4 is implemented, the Software Lock is not implemented. If it is not implemented, the behavior is as if it is unlocked. The Software Locks are controlled by EDLSR and EDLAR, PMLSR and PMLAR, and CTILSR and CTILAR. See Management registers and CoreSight compliance.

If FEAT_DoPD is implemented, Software Lock is not implemented by the architecturally-defined debug components in the Core power domain.

With the exception of these registers and the effect of the Software Lock, the behavior of the memory-mapped accesses is the same as for other accesses to the external debug interface.

Note

The recommended memory-mapped accesses to the external debug interface are not compatible with the memory-mapped interface defined in Armv7. In particular:

- The memory map is different.
- Memory-mapped accesses do not behave differently to Debug Access Port accesses when OSLSR_EL1, OS Lock Status Register.OSLK == 1, meaning that the OS Lock is locked.

The following sections give more information about these memory-mapped accesses:

- Register access permissions for memory-mapped accesses.
- Synchronization of memory-mapped accesses to external debug registers.

See also Supported access sizes.

E8.4.1 Register access permissions for memory-mapped accesses

It is IMPLEMENTATION DEFINED whether unprivileged memory-mapped accesses are allowed. Privileged software is responsible for controlling memory-mapped accesses using the MMU.

If FEAT_Debugv8p4 is implemented, the Security state view of a debug component is mapped into the physical memory for that Security state.

If FEAT_Debugv8p4 is implemented, the access permissions are different in each Security state, but Secure and Non-secure views of the debug components are identical. Arm recommends the views are located at the same address in the Secure and Non-secure physical address maps.

If memory-mapped accesses are made through an ADIv5 interface, the Debug Access Port can block the access using DBGSWENABLE. This is outside the scope of the Arm architecture. See *Arm® Debug Interface Architecture Specification ADIv5.0 to ADIv5.2*.

E8.4.1.1 Effect of the OPTIONAL Software Lock on memory-mapped access

For memory-mapped accesses, if other controls permit access to a register, the OPTIONAL Software Lock is implemented, and EDLSR.SLK, PMLSR.SLK, or CTILSR.SLK is set to 1, meaning the Software Lock is locked, then with the exception of the LAR itself:

- If other controls permit access to a register, then writes are ignored. That is:
  - Read/write (RW) registers become read-only, writes ignored (RO/WI).
  - Write-only (WO) registers become writes ignored (WI).

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Reads and writes have no side-effects. A side-effect is where a direct read or a direct write of a register creates an indirect write of the same or another register. When the Software Lock is locked, the indirect write does not occur.
- Writes to EDLAR, PMLAR, and CTILAR are unaffected.

This behavior must also apply to all IMPLEMENTATION DEFINED registers.

For example, if EDLSR.SLK is set to 1:

- EDSCR.{TXfull, TXU, ERR} are unchanged by a memory-mapped read from DBGDTRTX_EL0.
- EDSCR.{RXfull, RXO, ERR} are unchanged by a memory-mapped write to DBGDTRRX_EL0 that is ignored.
- EDSCR.{ITE, ITO, ERR} are unchanged by a memory-mapped write to EDITR that is ignored.
- OSLSR_EL1, OS Lock Status Register.OSLK is unchanged by a memory-mapped write to OSLAR_EL1 that is ignored.
- EDPCSR[63:32], EDCIDSR, and EDVIDSR are unchanged by a memory-mapped read from EDPCSR[31:0].

  Note

  Updating EDVIDSR, EDCIDSR, and EDPCSRhi are side-effects of reading EDPCSRlo, such that these registers contain the matching context for EDPCSRlo. The process that updates EDPCSRlo with PC samples is not a side-effect of the access. Reads of EDPCSRlo made when the Software Lock is locked can be used to profile software.

- PMPCSR[63:32], PMCID1SR/PMCID2SR, and PMVIDSR are unchanged by a memory-mapped read from PMPCSR[31:0].

  Note

  Updating PMVIDSR, PMCID1SR/PMCID2SR, and PMPCSR[31:0] are side-effects of reading PMPCSR[63:32], such that these registers contain the matching context for PMPCSR[63:32]. The process that updates PMPCSR[63:32] with PC samples is not a side-effect of the access. Reads of PMPCSR[63:32] made when the Software Lock is locked can be used to profile software.

- EDPRSR.{SDR, SPMAD, SDAD, SR, SPD} are unchanged by a memory-mapped read from EDPRSR.
- EDPRSR.SDAD is not set if an error response is returned due to a memory-mapped read or write of any debug register as the result of the value of the EDAD field.
- The CLAIM tags are unchanged by memory-mapped writes to DBGCLAIMSET_EL1 and DBGCLAIMCLR_EL1 which are ignored.

Similarly, if PMLSR.SLK is set to 1, then EDPRSR.SPMAD is not set if an error response is returned to a memory-mapped read or write of any Performance Monitors register due to the value of the EPMAD field.

E8.4.1.2 Behavior of a not permitted memory-mapped access

Where the architecture requires that an external debug interface access generates an error response, a memory-mapped access must also generate an error response. However, it is IMPLEMENTATION DEFINED how the error response is handled, as this depends on the system.

Arm recommends that the error is returned as either:

- A synchronous external Data Abort.
- An SError exception.

E8.4.2 Synchronization of memory-mapped accesses to external debug registers

The synchronization requirements for memory-mapped accesses to the external debug interface is described in Synchronization of changes to the external debug registers and Synchronization of memory-mapped registers.

The synchronization requirements between different routes to the external debug interface, that is, between Debug Access Port accesses and memory-mapped accesses are IMPLEMENTATION DEFINED.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E8.5 External debug interface register access permissions

Some external accesses to debug registers and Performance Monitor registers are not permitted and return an error response if:

- The Core power domain is powered down or is in low-power state where the registers cannot be accessed.
- OSLSR_EL1, OS Lock Status Register.OSLK == 1. The OS Lock is locked.
- FEAT_DoubleLock is implemented and `DoubleLockStatus()` == TRUE. The OS Double Lock is locked.
- The access is disabled by either the authentication interface or secure monitor.

Not all registers are affected in all of these cases. For more information, see External debug interface register access permissions summary.

E8.5.1 External debug over powerdown and locks

Accessing registers using the external debug interface is not possible when the Debug power domain is off. In this case, all accesses return an error.

External accesses to debug and Performance Monitors registers in the Core power domain are not permitted and return an error response if:

- The Core power domain is off or in low-power state where the registers cannot be accessed.
- OSLSR_EL1, OS Lock Status Register.OSLK == 1, meaning that the OS Lock is locked. This allows software to prevent external debugger modification of the registers while it saves and restores them over powerdown.
- FEAT_DoubleLock is implemented and `DoubleLockStatus()` == TRUE. This means that the OS Double Lock is locked. The OS Double Lock ensures that it is safe to remove Core power by forcing the debug interface to be quiescent.

If FEAT_DoubleLock is not implemented, the hardware must provide another method to safely remove Core power.

The OS Lock condition does not apply to the following debug registers:

- OSLAR_EL1. This means that an external debugger can override this lock.
- EDESR. This means that an external debugger can program a debug event for when software unlocks the OS Lock. See OS Unlock Catch debug event.
- The ID registers that describe the PE to the debugger.

See also Debug registers to save over powerdown.

E8.5.2 External access disabled

Accesses are further controlled by the external authentication interface. An untrusted external debugger cannot program the breakpoint and watchpoint registers to generate spurious debug exceptions. If external invasive debugging is not enabled, these external accesses to the registers are disabled.

The disable applies to:

- The DBGBVR<n>_EL1, DBGBCR<n>_EL1, DBGWVR<n>_EL1, and DBGWCR<n>_EL1 registers.
- From Armv8.2, the OSLAR_EL1 register.

If FEAT_Debugv8p4 is implemented, accesses from the external debug interface to these registers are not permitted if any of the following are true:

- The PE behaves as if the Security state is Secure, and the access is Non-secure.
- The access is not permitted by [**multi:sdcr?**].EDAD.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The `AllowExternalDebugAccess()` pseudocode function describes these accessibility rules.

PEs might also provide an OPTIONAL external debug interface to the Performance Monitor registers. The authentication interface and [**multi:sdcr?**] provide similar external access disable controls for those registers.

Note

Arm recommends that Secure software that is not making use of debug hardware does not lock out the external debug interface.

The `AllowExternalPMUAccess()` pseudocode function describes these accessibility rules.

E8.5.3 Behavior of a not permitted access

For an external debug interface access by a Debug Access Port, the Debug Access Port receives the error response and must signal this to the external debugger. For an ADIv5 implementation of a Debug Access Port, the error sets a sticky error flag in the Debug Access Port that the debugger can poll, and that suppresses further accesses until it is explicitly cleared.

When an error is returned because external access is disabled, and this is the highest priority error condition, a sticky error flag in EDPRSR is indirectly written to 1 as a side-effect of the access:

- For a debug register access when `AllowExternalDebugAccess()` == FALSE, EDPRSR.SDAD is indirectly written to 1.
- For Performance Monitor register access when `AllowExternalPMUAccess()` == FALSE, EDPRSR.SPMAD is indirectly written to 1.

The indirect write might not occur for a memory-mapped access to the external debug interface. For more information, see Register access permissions for memory-mapped accesses.

If no error is returned, or the error is returned because of a higher priority error condition, the flag in EDPRSR is unchanged.

See also Behavior of a not permitted memory-mapped access.

For more information, see *Arm® Debug Interface Architecture Specification*.

E8.5.4 External debug interface register access permissions summary

For accesses to:

- IMPLEMENTATION DEFINED registers, see IMPLEMENTATION DEFINED registers.
- OPTIONAL registers for CoreSight compliance, see Arm® Architecture Reference Manual, for A-profile architecture.
- Reserved, unallocated, or unimplemented registers, writes to read-only registers, and reads of write-only registers, see Reserved and unallocated registers.

For all other external debug interface, CTI, and Performance Monitor registers, Table E8-3, Table E8-4, Table E8-6 and Table F3-1, show the response of the PE to accesses by the external debug interface.

E8.5.5 IMPLEMENTATION DEFINED registers

Register access permissions for IMPLEMENTATION DEFINED debug, Performance Monitors, and CTI registers are IMPLEMENTATION DEFINED, other than as follows:

- If FEAT_DoPD is implemented, then all of the following apply:
  - All debug and Performance Monitors registers, including IMPLEMENTATION DEFINED registers, are implemented in the Core power domain and accesses return an error when the Core power domain is off or in a low-power state.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - CTI registers, including IMPLEMENTATION DEFINED registers, are implemented in the Debug power domain.
- When memory-mapped access to the external debug interface is supported, the additional constraints in Register access permissions for memory-mapped accesses apply to IMPLEMENTATION DEFINED registers.

E8.5.6 Reserved and unallocated registers

The default access requirements for reserved and unallocated registers are described in Access requirements for reserved and unallocated registers.

Note

Reads of WO and writes to RO refers to the default access permissions for a register. For example, when the SLK field is 1, meaning that the relevant registers become RO, a memory-mapped write to a RW register is ignored, and not treated as a reserved access.

The following reserved registers are RES0 in all conditions, other than when debug power is off:

- All reserved CTI registers.
- For the debug registers, and Performance Monitors registers, if the implementation is CoreSight architecture compliant, and either FEAT_DoPD is not implemented or the Core power domain is on, all reserved registers in the range `0xF00 - 0xFFC`. See Management register access permissions.

Otherwise, the architecture defines that:

1. If debug power is off, all register accesses, including reserved accesses, return an error.
2. For reserved debug registers and Performance Monitors registers, if FEAT_DoPD is implemented, and the Core power domain is off or in a low-power state, the response is an error. Otherwise, the response is a CONSTRAINED UNPREDICTABLE choice of error or RES0, when any of the following hold:

   Off The Core power domain is either completely off or in a low-power state in which the Core power domain registers cannot be accessed.

   DLK FEAT_DoubleLock is implemented and `DoubleLockStatus()` == TRUE. The OS Double Lock is locked.

   OSLK OSLSR_EL1, OS Lock Status Register.OSLK == 1. The OS Lock is locked.
3. In addition, for reserved debug registers in the address ranges `0x400 - 0x4FC` and `0x800 - 0x8FC`, the response is a CONSTRAINED UNPREDICTABLE choice of error or RES0 when conditions 1 or 2 do not apply and:

   EDAD `AllowExternalDebugAccess()` == FALSE. External debug is disabled.

   Note

   See also Behavior of a not permitted access.
4. In addition, for reserved Performance Monitors registers in the address ranges `0x000 - 0xEFC`, the response is a CONSTRAINED UNPREDICTABLE choice of error or RES0 when conditions 1 or 2 do not apply and:

   EPMAD `AllowExternalPMUAccess()` == FALSE. External Performance Monitor access is disabled.

   Note

   See also Behavior of a not permitted access.
5. For reads of WO locations, the response is a CONSTRAINED UNPREDICTABLE choice of error or RES0 when the architecture permits or requires a write to the location to return an error.
6. For writes of RO locations, the response is a CONSTRAINED UNPREDICTABLE choice of error or RES0 when the architecture permits or requires a read to the location to return an error.
7. For reads and writes of locations for features that are not implemented, the response is a CONSTRAINED UNPREDICTABLE choice of error or RES0 when the architecture permits or requires an access to the location to return an error if the feature is implemented.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E8.6 External debug interface registers

The external debug interface register map is described by:

- Summary of external debug registers.
- Cross-trigger interface registers.
- Performance Monitors external register views.

Table E8-2 Summary of external debug registers

Offset	Name	Description	Access
0x020	EDESR	External Debug Event Status Register	RW
0x024	EDECR	External Debug Execution Control Register	RW
0x030	EDWAR	External Debug Watchpoint Address Register	RO
0x080	DBGDTRRX_EL0	Debug Data Transfer Register, Receive	RW
0x084	EDITR	External Debug Instruction Transfer Register	WO
0x088	EDSCR	External Debug Status and Control Register	RW
0x08C	DBGDTRTX_EL0	Debug Data Transfer Register, Transmit	RW
0x090	EDRCR	External Debug Reserve Control Register	WO
0x094	EDACR	External Debug Auxiliary Control Register	RW
0x098	EDECCR	External Debug Exception Catch Control Register	RW
0x300	OSLAR_EL1	OS Lock Access Register	WO
0x310	EDPRCR	External Debug Power/Reset Control Register	RW
0x314	EDPRSR	External Debug Processor Status Register	RO
0x400 + (16 * n)	DBGBVR<n>_EL1	Debug Breakpoint Value Registers	RW
0x408 + (16 * n)	DBGBCR<n>_EL1	Debug Breakpoint Control Registers	RW
0x800 + (16 * n)	DBGWVR<n>_EL1	Debug Watchpoint Value Registers	RW
0x808 + (16 * n)	DBGWCR<n>_EL1	Debug Watchpoint Control Registers	RW
0xD00	MIDR_EL1	Main ID Register	RO
0xD20	EDPFR	External Debug Processor Feature Register	RO
0xD28	EDDFR	External Debug Feature Register	RO
0xD60	EDAA32PFR	External Debug Auxiliary Processor Feature Register	RO
0xF00	EDITCTRL	External Debug Integration mode Control register	RW
0xFA0	DBGCLAIMSET_EL1	Debug CLAIM Tag Set Register	RW
0xFA4	DBGCLAIMCLR_EL1	Debug CLAIM Tag Clear Register	RW
0xFA8	EDDEVAFF0	External Debug Device Affinity register 0	RO
0xFAC	EDDEVAFF1	External Debug Device Affinity register 1	RO
0xFB0	EDLAR	External Debug Lock Access Register	WO
0xFB4	EDLSR	External Debug Lock Status Register	RO
0xFB8	DBGAUTHSTATUS_EL1	Debug Authentication Status Register	RO
0xFBC	EDDEVARCH	External Debug Device Architecture Register	RO
0xFC0	EDDEVID2	External Debug Device ID register 2	RO

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Offset	Name	Description	Access
0xFC4	EDDEVID1	External Debug Device ID Register 1	RO
0xFC8	EDDEVID	External Debug Device ID register 0	RO
0xFCC	EDDEVTYPE	External Debug Device Type register	RO
0xFD0	EDPIDR4	External Debug Peripheral Identification Register 4	RO
0xFE0	EDPIDR0	External Debug Peripheral Identification Register 0	RO
0xFE4	EDPIDR1	External Debug Peripheral Identification Register 1	RO
0xFE8	EDPIDR2	External Debug Peripheral Identification Register 2	RO
0xFEC	EDPIDR3	External Debug Peripheral Identification Register 3	RO
0xFF0	EDCIDR0	External Debug Component Identification Register 0	RO
0xFF4	EDCIDR1	External Debug Component Identification Register 1	RO
0xFF8	EDCIDR2	External Debug Component Identification Register 2	RO
0xFFC	EDCIDR3	External Debug Component Identification Register 3	RO

E8.6.1 Access permissions for the External debug interface registers

Table E8-3 and Table E8-4 show the access permissions for the external debug interface registers in an Arm A-profile Debug implementation. The terms are defined as follows:

Domain This describes the power domain in which the register is logically implemented. Registers described as implemented in the Core power domain might be implemented in the Debug power domain, as long as they exhibit the required behavior.

If FEAT_DoPD is implemented, most External debug interface registers are in the Core power domain, as shown in Table E8-3.

If FEAT_DoPD is not implemented, most of the registers are in the Debug Power Domain, as shown in Table E8-4.

Conditions This lists the conditions under which the access is attempted.

To determine the access permissions for a register, read these columns from left to right, and stop at first column that lists the condition as being true.

The conditions are:

Off The Core power domain is completely off, or in low-power state. In these cases the Core power domain registers cannot be accessed, and if FEAT_DoPD is not implemented, EDPRSR.PU will read as 0.

Note

When the Core power domain is off, or in a low-power state, a debugger is permitted to access a debug register that is implemented in the external Debug power domain. When the Debug power domain is off, all accesses to the registers in the external Debug power domain return an error.

DLK FEAT_DoubleLock is implemented and `DoubleLockStatus()` == TRUE. The OS Double Lock is locked. If FEAT_DoPD is implemented, FEAT_DoubleLock is not implemented and so Table E8-3 does not include this column.

OSLK OSLSR_EL1, OS Lock Status Register.OSLK == 1. The OS Lock is locked.

EDAD `AllowExternalDebugAccess()` == FALSE. External debug access is disabled for the access. See also Behavior of a not permitted access.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EPMAD `AllowExternalPMUAccess()` == FALSE. Access to the external Performance Monitors is disabled for the access. See also Behavior of a not permitted access.

SLK The Software Lock is implemented and `SoftwareLockStatus()` == TRUE. This provides the modified default access permissions for OPTIONAL memory-mapped accesses to the external debug interface if the OPTIONAL Software Lock is locked for the access. See Register access permissions for memory-mapped accesses. If FEAT_DoPD is implemented, the Software Lock is not locked or not implemented, then this column is ignored.

Default This provides the default access permissions, if there are no conditions that prevent access to the register.

The access permissions are:

- This means that the default access permission applies. See the Default column, or the SLK column, if applicable.

RO This means that the register or field is read-only, and:

- Unless the register description states otherwise, a RO field in an RW register ignores writes.
- Where the `SLK` control makes a RW register RO, the register ignores writes.

RW This means that the register or field is read/write. Individual fields within the register might be RO or WO. See the relevant register description for details.

RC This means that a read of the register bit clears the field to 0.

WO This means that the register or field is write-only. Unless the register description states otherwise, a WO field in a RW register returns an UNKNOWN value on a read of the register.

WI This means that the register or field ignores writes.

IMP DEF This means that the access permissions are IMPLEMENTATION DEFINED.

DBGBVR<n>_EL1 and DBGWVR<n>_EL1, and if FEAT_Debugv8p9 is implemented, DBGBCR<n>_EL1 and DBGWCR<n>_EL1, are 64-bit registers mapped to pairs of 32-bit locations in the external debug interface. Doubleword accesses to these register are not guaranteed to be 64-bit single copy atomic. Software must take care before altering the values of DBGBVR<n>_EL1, DBGWVR<n>_EL1, DBGBCR<n>_EL1, and DBGWCR<n>_EL1 to prevent UNPREDICTABLE behavior. See Endianness and supported access sizes for more information.

If OPTIONAL memory-mapped access to the external debug interface is supported, there might be additional constraints on memory-mapped accesses. See Register access permissions for memory-mapped accesses.

For the reset values for the external debug interface registers, see Table E8-7.

Table E8-3 Access permissions for the external debug interface registers if FEAT_DoPD is implemented

		Domain	Conditions (priority from left to right)			
Offset	**Register**		**Off**	**OSLK**	**EDAD**	**Default**
`0x020`	EDESR	Core	Error	-	-	RW
`0x024`	EDECR	Core	Error	-	-	RW
`0x030`	EDWAR[31:0]	Core	Error	Error	-	RO
`0x034`	EDWAR[63:32]	Core	Error	Error	-	RO
`0x080`	DBGDTRRX_EL0	Core	Error	Error	-	RW
`0x084`	EDITR	Core	Error	Error	-	WO
`0x088`	EDSCR	Core	Error	Error	-	RW
`0x08C`	DBGDTRTX_EL0	Core	Error	Error	-	RW

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Offset	Register	Domain	Conditions (priority from left to right)			
			Off	OSLK	EDAD	Default
0x090	EDRCR	Core	Error	Error	-	WO
0x094	EDACR	Core	Error	IMP DEF	-	RW
0x098	EDECCR	Core	Error	Error	-	RW
0x300	OSLAR_EL1	Core	Error	-	Error	WO
0x310	EDPRCR	Core	Error	-	-	RW
0x314	EDPRSR	Core	Error	-	-	RO
0x400+16*n	DBGBVR<n>_EL1[31:0][a]	Core	Error	Error	Error	RW
0x404+16*n	DBGBVR<n>_EL1[63:32][a]	Core	Error	Error	Error	RW
0x408+16*n	DBGBCR<n>_EL1[31:0][a]	Core	Error	Error	Error	RW
0x40C+16*n	DBGBCR<n>_EL1[63:32] [a]	Core	Error	Error	Error	RW
0x800+16*n	DBGWVR<n>_EL1[31:0][a]	Core	Error	Error	Error	RW
0x804+16*n	DBGWVR<n>_EL1[63:32][a]	Core	Error	Error	Error	RW
0x808+16*n	DBGWCR<n>_EL1[31:0][a]	Core	Error	Error	Error	RW
0x80C+16*n	DBGWCR<n>_EL1[63:32] [a]	Core	Error	Error	Error	RW
0xD00	MIDR_EL1	Core	Error	-	-	RO
0xD20	EDPFR[31:0]	Core	Error	-	-	RO
0xD24	EDPFR[63:32]	Core	Error	-	-	RO
0xD28	EDDFR[31:0]	Core	Error	-	-	RO
0xD2C	EDDFR[63:32]	Core	Error	-	-	RO
0xD60	EDAA32PFR[31:0]	Core	Error	-	-	RO
0xD64	EDAA32PFR[63:32]	Core	Error	-	-	RO
0xFA0	DBGCLAIMSET_EL1	Core	Error	Error	-	RW
0xFA4	DBGCLAIMCLR_EL1	Core	Error	Error	-	RW
0xFA8	EDDEVAFF0	Core	Error	-	-	RO
0xFAC	EDDEVAFF1	Core	Error	-	-	RO
0xFB8	DBGAUTHSTATUS_EL1	Core	Error	-	-	RO
0xFC0	EDDEVID2	Core	Error	-	-	RO
0xFC4	EDDEVID1	Core	Error	-	-	RO
0xFC8	EDDEVID	Core	Error	-	-	RO

a. Implemented breakpoints and watchpoints only. *n* is the breakpoint or watchpoint number.

Table E8-4 Access permissions for the external debug interface registers if FEAT_DoPD is not implemented

Offset	Register	Domain	Conditions (priority from left to right)				
			Off	DLK	OSLK	Default	SLK
0x020	EDESR	Core	Error	Error	-	RW	RO
0x024	EDECR	Debug	-	-	-	RW	RO
0x028	EDSCR2	Core	Error	Error	Error	RW	RO

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Offset	Register	Domain	Conditions (priority from left to right) Off	DLK	OSLK	Default	SLK
0x030	EDWAR[31:0]	Core	Error	Error	Error	RO	-
0x034	EDWAR[63:32]	Core	Error	Error	Error	RO	-
0x080	DBGDTRRX_EL0	Core	Error	Error	Error	RW	RO
0x084	EDITR	Core	Error	Error	Error	WO	WI
0x088	EDSCR	Core	Error	Error	Error	RW	RO
0x08C	DBGDTRTX_EL0	Core	Error	Error	Error	RW	RO
0x090	EDRCR	Core	Error	Error	Error	WO	WI
0x094	EDACR	IMP DEF	IMP DEF	IMP DEF	IMP DEF	RW	RO
0x098	EDECCR	Core	Error	Error	Error	RW	RO
0x300	OSLAR_EL1	Core	Error	Error	-	WO	WI
0x310	EDPRCR	Core and Debug[a]	-	-	-	RW	RO
0x314	EDPRSR	Core and Debug[a]	-	-	-	RO	RO
0x400+16*n	DBGBVR<n>_EL1[31:0][b]	Core	Error	Error	Error	RW	RO
0x404+16*n	DBGBVR<n>_EL1[63:32][b]	Core	Error	Error	Error	RW	RO
0x408+16*n	DBGBCR<n>_EL1[31:0][b]	Core	Error	Error	Error	RW	RO
0x40C+16*n	DBGBCR<n>_EL1[63:32][b]	Core	Error	Error	Error	RW	RO
0x800+16*n	DBGWVR<n>_EL1[31:0][b]	Core	Error	Error	Error	RW	RO
0x804+16*n	DBGWVR<n>_EL1[63:32][b]	Core	Error	Error	Error	RW	RO
0x808+16*n	DBGWCR<n>_EL1[31:0][b]	Core	Error	Error	Error	RW	RO
0x80C+16*n	DBGWCR<n>_EL1[63:32][b]	Core	Error	Error	Error	RW	RO
0xD00	MIDR_EL1	IMP DEF	IMP DEF[c]	IMP DEF[c]	-	RO	RO
0xD20	EDPFR[31:0]	IMP DEF	IMP DEF[c]	IMP DEF[c]	-	RO	RO
0xD24	EDPFR[63:32]	IMP DEF	IMP DEF[c]	IMP DEF[c]	-	RO	RO
0xD28	EDDFR[31:0]	IMP DEF	IMP DEF[c]	IMP DEF[c]	-	RO	RO
0xD2C	EDDFR[63:32]	IMP DEF	IMP DEF[c]	IMP DEF[c]	-	RO	RO
0xD60	EDAA32PFR[31:0]	IMP DEF	IMP DEF[c]	IMP DEF[c]	-	RO	RO
0xD64	EDAA32PFR[63:32]	IMP DEF	IMP DEF[c]	IMP DEF[c]	-	RO	RO
0xFA0	DBGCLAIMSET_EL1	Core	Error	Error	Error	RW	RO
0xFA4	DBGCLAIMCLR_EL1	Core	Error	Error	Error	RW	RO
0xFA8	EDDEVAFF0	Debug	-	-	-	RO	RO
0xFAC	EDDEVAFF1	Debug	-	-	-	RO	RO
0xFB8	DBGAUTHSTATUS_EL1	Debug	-	-	-	RO	RO
0xFC0	EDDEVID2	Debug	-	-	-	RO	RO
0xFC4	EDDEVID1	Debug	-	-	-	RO	RO
0xFC8	EDDEVID	Debug	-	-	-	RO	RO

a. Some bits are in the Debug power domain and some bits are in the Core power domain. See register field descriptions for information.

b. Implemented breakpoints and watchpoints only. *n* is the breakpoint or watchpoint number.

c. It is IMPLEMENTATION DEFINED whether an error is returned. See External debug over powerdown and locks. If no error is returned, the access is permitted.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E8.7 Cross-trigger interface registers

The Embedded Cross-Trigger Interface, CTI, is located within its own block of the external debug memory map. There must be one such block for each PE.

If the CTI of a PE does not implement the CTIDEVAFF0 or CTIDEVAFF1 registers it must be located 64KB above the debug registers in the external debug interface.

When FEAT_Debugv8p4 is implemented, each debug component has a Secure and Non-secure view. The Secure view of a debug component is mapped into Secure physical memory and the Non-secure view of a debug component is mapped into Non-secure memory. Apart from access conditions, the Non-secure and Secure views of the debug components are identical.

Table E8-5 shows the CTI register map.

Table E8-5 Summary of the Embedded Cross-Trigger Interface, CTI, registers

Offset	Name	Description	Access
0x000	CTICONTROL	CTI Control register	RW
0x010	CTIINTACK	CTI Output Trigger Acknowledge register	WO
0x014	CTIAPPSET	CTI Application Trigger Set register	RW
0x018	CTIAPPCLEAR	CTI Application Trigger Clear register	WO
0x01C	CTIAPPPULSE	CTI Application Pulse register	WO
0x020 + (4 * n)	CTIINEN<n>	CTI Input Trigger to Output Channel Enable registers	RW
0x0A0 + (4 * n)	CTIOUTEN<n>	CTI Input Channel to Output Trigger Enable registers	RW
0x130	CTITRIGINSTATUS	CTI Trigger In Status register	RO
0x134	CTITRIGOUTSTATUS	CTI Trigger Out Status register	RO
0x138	CTICHINSTATUS	CTI Channel In Status register	RO
0x13C	CTICHOUTSTATUS	CTI Channel Out Status register	RO
0x140	CTIGATE	CTI Channel Gate Enable register	RW
0x144	ASICCTL	CTI External Multiplexer Control register	RO
0x150	CTIDEVCTL	CTI Device Control register	RW
0xF00	CTIITCTRL	CTI Integration mode Control register	RW
0xFA0	CTICLAIMSET	CTI CLAIM Tag Set register	RW
0xFA4	CTICLAIMCLR	CTI CLAIM Tag Clear register	RW
0xFA8	CTIDEVAFF0	CTI Device Affinity register 0	RO
0xFAC	CTIDEVAFF1	CTI Device Affinity register 1	RO
0xFB0	CTILAR	CTI Lock Access Register	WO
0xFB4	CTILSR	CTI Lock Status Register	RO
0xFB8	CTIAUTHSTATUS	CTI Authentication Status register	RO
0xFBC	CTIDEVARCH	CTI Device Architecture register	RO
0xFC0	CTIDEVID2	CTI Device ID register 2	RO
0xFC4	CTIDEVID1	CTI Device ID register 1	RO
0xFC8	CTIDEVID	CTI Device ID register 0	RO
0xFCC	CTIDEVTYPE	CTI Device Type register	RO
0xFD0	CTIPIDR4	CTI Peripheral Identification Register 4	RO

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Offset	Name	Description	Access
0xFE0	CTIPIDR0	CTI Peripheral Identification Register 0	RO
0xFE4	CTIPIDR1	CTI Peripheral Identification Register 1	RO
0xFE8	CTIPIDR2	CTI Peripheral Identification Register 2	RO
0xFEC	CTIPIDR3	CTI Peripheral Identification Register 3	RO
0xFF0	CTICIDR0	CTI Component Identification Register 0	RO
0xFF4	CTICIDR1	CTI Component Identification Register 1	RO
0xFF8	CTICIDR2	CTI Component Identification Register 2	RO
0xFFC	CTICIDR3	CTI Component Identification Register 3	RO

Table E8-6 shows the access permissions for the CTI registers in an Arm A-profile Debug implementation. For a definition of the terms used, see External debug interface registers.

Table E8-6 Access permissions for the CTI registers

			Conditions (priority from left to right)					
Offset	Register	Domain	Off	DLK	OSLK	EDAD	Default	SLK
0x000	CTICONTROL	Debug	-	-	-	-	RW	RO
0x010	CTIINTACK	Debug	-	-	-	-	WO	WI
0x014	CTIAPPSET	Debug	-	-	-	-	RW	RO
0x018	CTIAPPCLEAR	Debug	-	-	-	-	WO	WI
0x01C	CTIAPPPULSE	Debug	-	-	-	-	WO	WI
0x020+4×n	CTIINEN<n>	Debug	-	-	-	-	RW	RO
0x0A0+4×n	CTIOUTEN<n>	Debug	-	-	-	-	RW	RO
0x130	CTITRIGINSTATUS	Debug	-	-	-	-	RO	RO
0x134	CTITRIGOUTSTATUS	Debug	-	-	-	-	RO	RO
0x138	CTICHINSTATUS	Debug	-	-	-	-	RO	RO
0x13C	CTICHOUTSTATUS	Debug	-	-	-	-	RO	RO
0x140	CTIGATE	Debug	-	-	-	-	RW	RO
0xFC0	CTIDEVID2	Debug	-	-	-	-	RO	RO
0xFC4	CTIDEVID1	Debug	-	-	-	-	RO	RO
0xFC8	CTIDEVID	Debug	-	-	-	-	RO	RO

For the reset values of the CTI registers, see Table E8-8.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E8.8 External debug register resets

Each register or field has a defined reset domain:

- Registers and fields in the Warm reset domain are also reset by a Cold reset and unchanged by an External Debug reset that is not coincident with a Cold reset or a Warm reset.
- Registers and fields in the Cold reset domain are unchanged by a Warm reset or an External Debug reset that is not coincident with a Cold reset.
- Registers and fields in the External Debug reset domain are unchanged by a Cold reset or a Warm reset that is not coincident with an External Debug reset.

A reset might change the value of a register. Specific rules apply to the observability of registers in the External Debug reset domain by indirect reads from the Core power domain when an External Debug reset is asserted without a coincident Cold reset. For more information, see Synchronization of changes to the external debug registers.

Table E8-2 and Table E8-5 show the external debug register and CTI register resets. For other debug registers, see Management register resets.

Note

By reference to Figure E6-1 the power domain can be deduced from the reset domain.

Table E8-2 and Table E8-5 do not include:

- Read-only identification registers, such as Processor ID Registers and PMCFGR, that have a fixed value from reset.
- Read-only status registers, such as EDSCR.RW, that are evaluated each time the register is read and that have no meaningful reset value.
- Write-only registers, such as EDRCR, that only have an effect on writes, and have no meaningful reset value.
- Read/write registers, such as breakpoint and watchpoint registers, and EDPRCR.CORENPDRQ, that alias other registers. The reset values are described by the descriptions of those other registers.
- IMPLEMENTATION DEFINED registers. The reset values and reset domains of these registers are also IMPLEMENTATION DEFINED and might be UNKNOWN.

All other fields in the registers are set to an IMPLEMENTATION DEFINED value that can be UNKNOWN. The register is in the specified reset domain.

Note

An IMPLEMENTATION DEFINED reset value, which can be UNKNOWN, means that hardware is not required to reset the register on the specified reset, but software must not rely on the register being preserved over reset.

Table E8-7 Summary of external debug register resets, debug registers

Register	Reset domain	Field	Value	Description
DBGPRCR_EL1	Cold into AArch64 state	CORENPDRQ	The value of the powerup request[a]	Debug Power Control Register.
EDESR, if FEAT_DoPD is not implemented	Warm	SS	EDECR.SS	Halting Step debug event pending
		RC	EDECR.RCE	Reset Catch debug event pending
		OSUC	0	OS Unlock Catch debug event pending
EDESR, if FEAT_DoPD is implemented	Cold	SS	0	Halting Step debug event pending
	Warm	RC	CTIDEVCTL.RCE	Reset Catch debug event pending

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Reset domain	Field	Value	Description
		OSUC	0	OS Unlock Catch debug event pending
EDECR, if FEAT_DoPD is implemented	Cold	SS	0	Halting Step debug event enable
EDECR, if FEAT_DoPD is not implemented	External debug	SS	0	Halting Step debug event enable
		RCE	0	Reset Catch debug event enable
		OSUCE	0	OS Unlock Catch debug event enable
EDWAR	Cold	-	-	All fields
EDSCR	Cold	RXfull	0	DTRRX register full
		TXfull	0	DTRTX register full
		RXO	0	DTRRX overrun
		TXU	0	DTRTX underrun
		INTdis	0	Interrupt disable
		TDA	0	Trap debug register accesses to Debug state
		MA	0	Memory access mode in Debug state
		HDE	0	Halting debug mode enable
		ERR	0	Cumulative error flag
EDECCR	Cold	NSE[2:1]	`0b00`	Coarse-grained Non-secure Exception Catch
		SE[3,1]	`0b00`	Coarse-grained Secure Exception Catch
EDPRCR, if FEAT_DoPD is implemented	Cold	-	-	-
EDPRCR, if FEAT_DoPD is not implemented	External debug	COREPURQ[b]	-	Core powerup request
EDPRSR	Warm	SDR	-	Sticky debug restart
	Cold	SPMAD	0	Sticky EPMAD error
		SDAD	0	Sticky EDAD error
	Warm	SR	1	Sticky reset status
	Cold	SPD	1	Sticky powerdown status

a. If FEAT_DoPD is not implemented, the powerup request is the EDPRCR.COREPURQ control bit.

b. If FEAT_DoPD is not implemented, on a Cold reset into AArch64 state, DBGPRCR_EL1.CORENPDRQ resets to the value of EDPRCR.COREPURQ. If an External Debug reset and a Cold reset coincide, both EDPRCR.COREPURQ and the CORENPDRQ field of the appropriate System register are reset to 0.

Table E8-5 shows the reset values for the CTI registers

Table E8-8 Summary of external debug register resets, CTI registers

Register	Reset domain	Field	Value	Description
CTICONTROL	External debug	GLBEN	0	CTI global enable

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Reset domain	Field	Value	Description
CTIDEVCTL	External debug	RCE	0	If FEAT_DoPD is implemented, Reset Catch debug event enable
		OSUCE	0	If FEAT_DoPD is implemented, OS Unlock Catch debug event enable
CTIAPPSET	External debug	-	-	All fields
CTIINEN<n>	External debug	-	-	All fields
CTIOUTEN<n>	External debug	-	-	All fields
CTIGATE	External debug	-	-	All fields
ASICCTL	IMPLEMENTATION DEFINED	-	IMPLEMENTATION DEFINED	All of register

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter E9
External Debug Register Descriptions

This chapter provides a description of the external debug registers. It contains the following sections:

- About the external debug registers.
- External debug registers.
- External trace registers.
- External Trace Buffer registers.
- Cross-Trigger Interface registers.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.1 About the external debug registers

The following sections describe the registers that are accessible through the external debug interface:

- External debug registers.
- External trace registers.
- Cross-Trigger Interface registers.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.2 External debug registers

This section describes the debug registers that are accessible through the external debug interface and are used for external debug.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.2.1 DBGAUTHSTATUS_EL1, Debug Authentication Status Register

The DBGAUTHSTATUS_EL1 characteristics are:

Purpose

Provides information about the state of the IMPLEMENTATION DEFINED authentication interface for debug.

Configuration

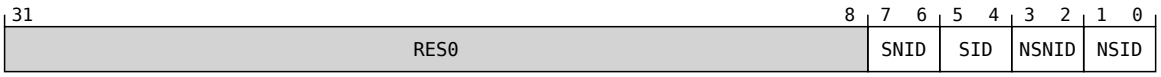
When FEAT_DoPD is implemented, DBGAUTHSTATUS_EL1 is in the Core power domain. Otherwise, DBGAUTHSTATUS_EL1 is in the Debug power domain

External register DBGAUTHSTATUS_EL1 bits [31:0] are architecturally mapped to AArch64 System register DBGAUTHSTATUS_EL1[31:0].

Attributes

DBGAUTHSTATUS_EL1 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

SNID, bits [7:6]

When FEAT_Debugv8p4 is implemented:

Secure non-invasive debug.

This field has the same value as DBGAUTHSTATUS_EL1.SID.

Otherwise:

Secure non-invasive debug.

SNID	Meaning
0b00	Secure state is not implemented.
0b10	Implemented and disabled. ExternalSecureNoninvasiveDebugEnabled() == FALSE.
0b11	Implemented and enabled. ExternalSecureNoninvasiveDebugEnabled() == TRUE.

All other values are reserved.

SID, bits [5:4]

Secure invasive debug.

SID	Meaning
0b00	Secure state is not implemented.
0b10	Implemented and disabled. ExternalSecureInvasiveDebugEnabled() == FALSE.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b11	Implemented and enabled. ExternalSecureInvasiveDebugEnabled() == TRUE.

All other values are reserved.

NSNID, bits [3:2]

When FEAT_Debugv8p4 is implemented:

Non-secure non-invasive debug.

NSNID	Meaning
0b00	Non-secure state is not implemented.
0b11	Implemented and enabled. ExternalNoninvasiveDebugEnabled() == TRUE.

All other values are reserved.

Otherwise:

Non-secure non-invasive debug.

NSNID	Meaning
0b00	Non-secure state is not implemented.
0b10	Implemented and disabled. ExternalNoninvasiveDebugEnabled() == FALSE.
0b11	Implemented and enabled. ExternalNoninvasiveDebugEnabled() == TRUE.

All other values are reserved.

NSID, bits [1:0]

Non-secure invasive debug.

NSID	Meaning
0b00	Non-secure state is not implemented.
0b10	Implemented and disabled. ExternalInvasiveDebugEnabled() == FALSE.
0b11	Implemented and enabled. ExternalInvasiveDebugEnabled() == TRUE.

All other values are reserved.

Accessing DBGAUTHSTATUS_EL1

DBGAUTHSTATUS_EL1 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xFB8	DBGAUTHSTATUS_EL1

Accessible as follows:

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.2.2 DBGBCR<n>_EL1, Debug Breakpoint Control Registers, n = 0 - 63

The DBGBCR<n>_EL1 characteristics are:

Purpose

Holds control information for a breakpoint. Forms breakpoint n together with value register DBGBVR<n>_EL1.

Configuration

DBGBCR<n>_EL1 is in the Core power domain

If breakpoint n is not implemented, then accesses to this register are:

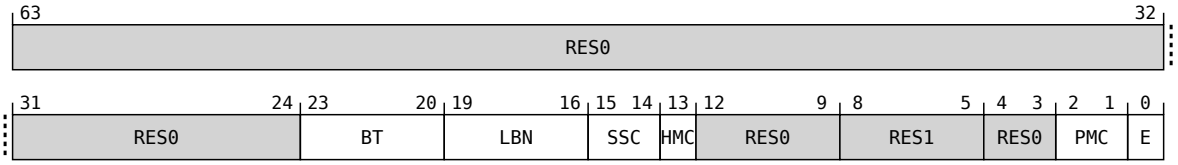
- RES0 when IsCorePowered() && !DoubleLockStatus() && !OSLockStatus() && AllowExternalDebugAccess().
- A CONSTRAINED UNPREDICTABLE choice of RES0 or ERROR otherwise.

External register DBGBCR<n>_EL1 bits [31:0] are architecturally mapped to AArch64 System register DBGBCR<n>_EL1[31:0].

Attributes

DBGBCR<n>_EL1 is a 64-bit register.

Field descriptions



When the E field is zero, all the other fields in the register are ignored.

Bits [63:24]

Reserved, RES0.

BT, bits [23:20]

Breakpoint Type.

Specifies breakpoint type.

BT	Meaning	Applies when
0b0000	Unlinked instruction address match. DBGBVR<n>_EL1 is the address of an instruction.	
0b0001	Linked instruction address match. As 0b0000, but linked to a breakpoint that has linking enabled.	
0b0010	Unlinked Context ID match. If the Effective value of HCR_EL2.E2H is 1 and either the PE is executing at EL0 with HCR_EL2.TGE set to 1 or the PE is executing at EL2, then DBGBVR<n>_EL1.ContextID is compared against CONTEXTIDR_EL2. Otherwise, DBGBVR<n>_EL1.ContextID is compared against CONTEXTIDR_EL1.	breakpoint n is context-aware
0b0011	As 0b0010, with linking enabled.	breakpoint n is context-aware

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

BT	Meaning	Applies when
0b0110	Unlinked CONTEXTIDR_EL1 match. DBGBVR<n>_EL1.ContextID is a Context ID compared against CONTEXTIDR_EL1.	FEAT_Debugv8p1 is implemented and breakpoint n is context-aware
0b0111	As 0b0110, with linking enabled.	FEAT_Debugv8p1 is implemented and breakpoint n is context-aware
0b1000	Unlinked VMID match. DBGBVR<n>_EL1.VMID is a VMID compared against VTTBR_EL2.VMID.	breakpoint n is context-aware
0b1001	As 0b1000, with linking enabled.	breakpoint n is context-aware
0b1010	Unlinked VMID and Context ID match. DBGBVR<n>_EL1.ContextID is a Context ID compared against CONTEXTIDR_EL1, and DBGBVR<n>_EL1.VMID is a VMID compared against VTTBR_EL2.VMID.	breakpoint n is context-aware
0b1011	As 0b1010, with linking enabled.	breakpoint n is context-aware
0b1100	Unlinked CONTEXTIDR_EL2 match. DBGBVR<n>_EL1.ContextID2 is a Context ID compared against CONTEXTIDR_EL2.	FEAT_Debugv8p1 is implemented and breakpoint n is context-aware
0b1101	As 0b1100, with linking enabled.	FEAT_Debugv8p1 is implemented and breakpoint n is context-aware
0b1110	Unlinked Full Context ID match. DBGBVR<n>_EL1.ContextID is compared against CONTEXTIDR_EL1, and DBGBVR<n>_EL1.ContextID2 is compared against CONTEXTIDR_EL2.	FEAT_Debugv8p1 is implemented and breakpoint n is context-aware
0b1111	As 0b1110, with linking enabled.	FEAT_Debugv8p1 is implemented and breakpoint n is context-aware

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

LBN, bits [19:16]

Linked Breakpoint Number.

For Linked address matching breakpoints, specifies the index of the breakpoint linked to.

For all other breakpoint types, this field is ignored and reads of the register return an UNKNOWN value.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

SSC, bits [15:14]

Security state control. Determines the Security states under which a Breakpoint debug event for breakpoint n is generated. This field must be interpreted along with the HMC and PMC fields, and there are constraints on the permitted values of the {HMC, SSC, PMC} fields. For more information, including the effect of programming the fields to a reserved set of values, see 'Reserved DBGBCR<n>_EL1.{SSC, HMC, PMC} values'.

For more information on the operation of the SSC, HMC, and PMC fields, see 'Execution conditions for which a breakpoint generates Breakpoint exceptions'.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

HMC, bit [13]

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Higher mode control. Determines the debug perspective for deciding when a Breakpoint debug event for breakpoint n is generated. This field must be interpreted along with the SSC and PMC fields, and there are constraints on the permitted values of the {HMC, SSC, PMC} fields. For more information see DBGBCR<n>_EL1.SSC description.

For more information on the operation of the SSC, HMC, and PMC fields, see 'Execution conditions for which a breakpoint generates Breakpoint exceptions'.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [12:9]

Reserved, RES0.

Bits [8:5]

Reserved, RES1.

Bits [4:3]

Reserved, RES0.

PMC, bits [2:1]

Privilege mode control. Determines the Exception level or levels at which a Breakpoint debug event for breakpoint n is generated. This field must be interpreted along with the SSC and HMC fields, and there are constraints on the permitted values of the {HMC, SSC, PMC} fields. For more information see the DBGBCR<n>_EL1.SSC description.

For more information on the operation of the SSC, HMC, and PMC fields, see 'Execution conditions for which a breakpoint generates Breakpoint exceptions'.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

E, bit [0]

Enable breakpoint n.

E	Meaning
0b0	Breakpoint n disabled.
0b1	Breakpoint n enabled.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing DBGBCR<n>_EL1

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalDebugAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

This register is 32-bits wide and offset 0x40C + (16 * n) is reserved.

DBGBCR<n>_EL1 can be accessed through the external debug interface:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Offset	Instance
Debug	`0x408 + (16 * n)`	DBGBCR<n>_EL1

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), or !AllowExternalDebugAccess(addrdesc), accesses to this register return an ERROR.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.2.3 DBGBVR<n>_EL1, Debug Breakpoint Value Registers, n = 0 - 63

The DBGBVR<n>_EL1 characteristics are:

Purpose

Holds a virtual address, or a VMID and/or a context ID, for use in breakpoint matching. Forms breakpoint n together with control register DBGBCR<n>_EL1.

Configuration

DBGBVR<n>_EL1 is in the Core power domain

How this register is interpreted depends on the value of DBGBCR<n>_EL1.BT.

- When DBGBCR<n>_EL1.BT is `0b0x0x`, this register holds a virtual address.
- When DBGBCR<n>_EL1.BT is `0b001x`, `0b011x`, or `0b110x`, this register holds a Context ID.
- When DBGBCR<n>_EL1.BT is `0b100x`, this register holds a VMID.
- When DBGBCR<n>_EL1.BT is `0b101x`, this register holds a VMID and a Context ID.
- When DBGBCR<n>_EL1.BT is `0b111x`, this register holds two Context ID values.

For other values of DBGBCR<n>_EL1.BT, this register is RES0.

If breakpoint n is not implemented, then accesses to this register are:

- RES0 when IsCorePowered() && !DoubleLockStatus() && !OSLockStatus() && AllowExternalDebugAccess().
- A CONSTRAINED UNPREDICTABLE choice of RES0 or ERROR otherwise.

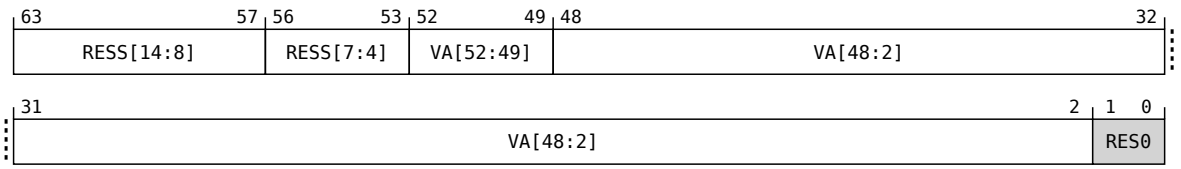
External register DBGBVR<n>_EL1 bits [63:0] are architecturally mapped to AArch64 System register DBGBVR<n>_EL1[63:0].

Attributes

DBGBVR<n>_EL1 is a 64-bit register.

Field descriptions

When DBGBCR<n>_EL1.BT IN {'`0x0`x'}:



RESS[14:8], bits [63:57]

Reserved, Sign extended. Software must treat this field as RES0 if the most significant bit of VA is 0 or RES0, and as RES1 if the most significant bit of VA is 1.

Hardware always ignores the value of these bits and it is IMPLEMENTATION DEFINED whether:

- The bits are hardwired to a copy of the most significant bit of VA, meaning writes to these bits are ignored, and reads to the bits always return the hardwired value.
- The value in those bits can be written, and reads will return the last value written. The value held in those bits is ignored by hardware.

RESS[7:4], bits [56:53]

Extension to RESS[14:8]. For more information, see RESS[14:8].

Bits [52:49]

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_LVA is implemented:

VA[52:49], bits [52:49]

Extension to VA[48:2]. For more information, see VA[48:2].

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

RESS[3:0], bits [52:49]

Extension to RESS[14:8]. For more information, see RESS[14:8].

VA[48:2], bits [48:2]

If the address is being matched in an AArch64 stage 1 translation regime:

- This field contains bits[48:2] of the address for comparison.
- When FEAT_LVA is implemented, VA[52:49] forms the upper part of the address value. If FEAT_LVA is not implemented, bits [52:49] are part of the RESS field.

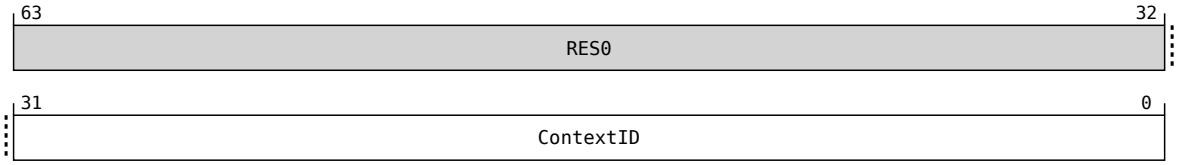
The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [1:0]

Reserved, RES0.

When DBGBCR<n>_EL1.BT IN {'001x'}:



Bits [63:32]

Reserved, RES0.

ContextID, bits [31:0]

Context ID value for comparison.

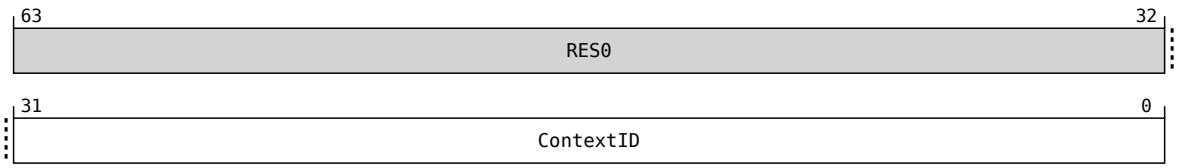
The value is compared against the following:

- CONTEXTIDR_EL1 when the PE is executing at AArch64.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

When DBGBCR<n>_EL1.BT IN {'011x'} and FEAT_Debugv8p1 is implemented:



Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [63:32]

Reserved, RES0.

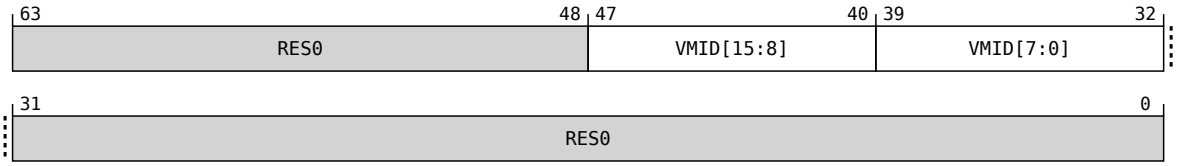
ContextID, bits [31:0]

Context ID value for comparison against CONTEXTIDR_EL1.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

When DBGBCR<n>_EL1.BT IN {'100x'}:



Bits [63:48]

Reserved, RES0.

VMID[15:8], bits [47:40]

When FEAT_VMID16 is implemented and VTCR_EL2.VS == '1':

Extension to VMID[7:0]. For more information, see DBGBVR<n>_EL1.VMID[7:0].

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

VMID[7:0], bits [39:32]

VMID value for comparison.

The VMID is 8 bits when any of the following are true:

- VTCR_EL2.VS is 0.
- FEAT_VMID16 is not implemented.

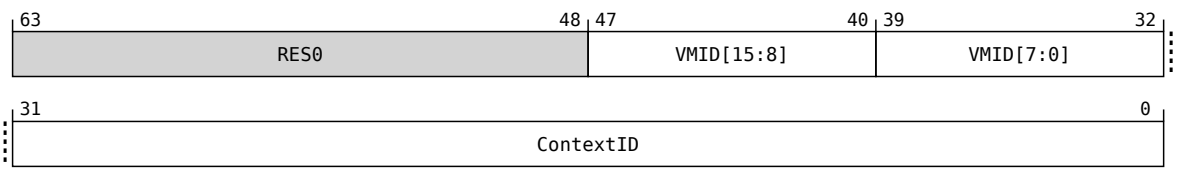
The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [31:0]

Reserved, RES0.

When DBGBCR<n>_EL1.BT IN {'101x'}:



Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [63:48]

Reserved, RES0.

VMID[15:8], bits [47:40]

When FEAT_VMID16 is implemented and VTCR_EL2.VS == '1':

Extension to VMID[7:0]. For more information, see DBGBVR<n>_EL1.VMID[7:0].

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

VMID[7:0], bits [39:32]

VMID value for comparison.

The VMID is 8 bits when any of the following are true:

- VTCR_EL2.VS is 0.
- FEAT_VMID16 is not implemented.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

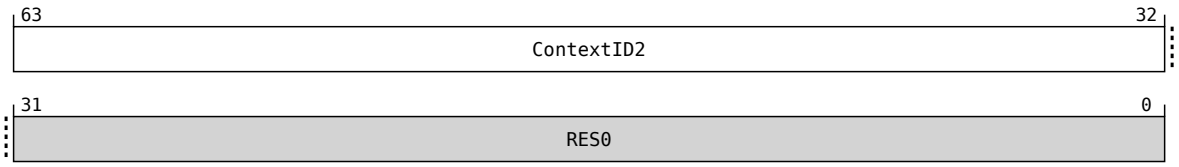
ContextID, bits [31:0]

Context ID value for comparison against CONTEXTIDR_EL1.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

When DBGBCR<n>_EL1.BT IN {'110x'} and FEAT_Debugv8p1 is implemented:



ContextID2, bits [63:32]

Context ID value for comparison against CONTEXTIDR_EL2.

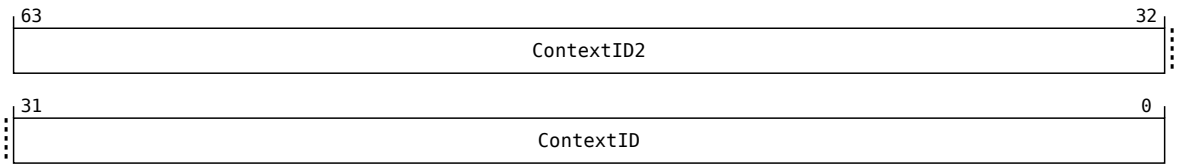
The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [31:0]

Reserved, RES0.

When DBGBCR<n>_EL1.BT IN {'111x'} and FEAT_Debugv8p1 is implemented:



Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ContextID2, bits [63:32]

Context ID value for comparison against CONTEXTIDR_EL2.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

ContextID, bits [31:0]

Context ID value for comparison against CONTEXTIDR_EL1.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing DBGBVR<n>_EL1

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalDebugAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

DBGBVR<n>_EL1 can be accessed through the external debug interface:

Component	Offset	Instance	Range
Debug	`0x400 + (16 * n)`	DBGBVR<n>_EL1	63:0

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), or !AllowExternalDebugAccess(addrdesc), accesses to this register return an ERROR.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.2.4 DBGCLAIMCLR_EL1, Debug CLAIM Tag Clear Register

The DBGCLAIMCLR_EL1 characteristics are:

Purpose

Used by software to read the values of the CLAIM tag bits, and to clear CLAIM tag bits to 0.

The architecture does not define any functionality for the CLAIM tag bits.

Note

CLAIM tags are typically used for communication between the debugger and target software.

Used in conjunction with the DBGCLAIMSET_EL1 register.

Configuration

DBGCLAIMCLR_EL1 is in the Core power domain

An implementation must include eight CLAIM tag bits.

External register DBGCLAIMCLR_EL1 bits [31:0] are architecturally mapped to AArch64 System register DBGCLAIMCLR_EL1[31:0].

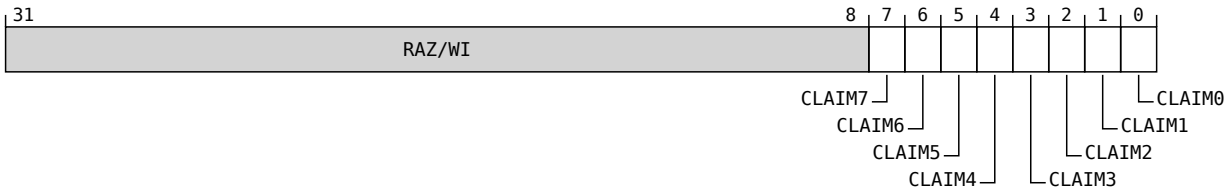
External register DBGCLAIMCLR_EL1 bits [31:0] are architecturally mapped to AArch64 System register DBGCLAIMSET_EL1[31:0].

External register DBGCLAIMCLR_EL1 bits [31:0] are architecturally mapped to External register DBGCLAIMSET_EL1[31:0].

Attributes

DBGCLAIMCLR_EL1 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RAZ/WI.

CLAIM<m> , bits [m], for m = 7 to 0

Claim Tag Clear. Indicates the current status of Claim Tag bit <m>, and is used to clear Claim Tag bit <m> to 0.

CLAIM<m>	Meaning
0b0	On a read: Claim Tag bit <m> is not set. On a write: Ignored.
0b1	On a read: Claim Tag bit <m> is set. On a write: Clear Claim tag bit <m> to 0.

The reset behavior of this field is:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Cold reset, this field resets to `'0'`.

Access to this field is W1C.

Accessing DBGCLAIMCLR_EL1

DBGCLAIMCLR_EL1 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0xFA4`	DBGCLAIMCLR_EL1

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), accesses to this register return an ERROR.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.2.5 DBGCLAIMSET_EL1, Debug CLAIM Tag Set Register

The DBGCLAIMSET_EL1 characteristics are:

Purpose

Used by software to set the CLAIM tag bits to 1.

The architecture does not define any functionality for the CLAIM tag bits.

Note

CLAIM tags are typically used for communication between the debugger and target software.

Used in conjunction with the DBGCLAIMCLR_EL1 register.

Configuration

DBGCLAIMSET_EL1 is in the Core power domain

An implementation must include eight CLAIM tag bits.

External register DBGCLAIMSET_EL1 bits [31:0] are architecturally mapped to AArch64 System register DBGCLAIMSET_EL1[31:0].

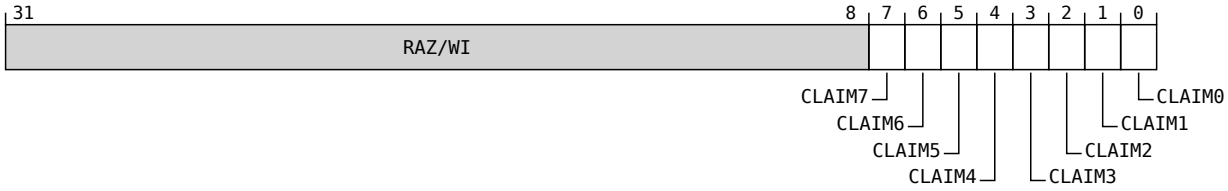
External register DBGCLAIMSET_EL1 bits [31:0] are architecturally mapped to AArch64 System register DBGCLAIMCLR_EL1[31:0].

External register DBGCLAIMSET_EL1 bits [31:0] are architecturally mapped to External register DBGCLAIMCLR_EL1[31:0].

Attributes

DBGCLAIMSET_EL1 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RAZ/WI.

CLAIM<m> , bits [m], for m = 7 to 0

Claim Tag Set. Used to set Claim Tag bit <m> to 1.

CLAIM<m>	Meaning
0b0	On a write: Ignored.
0b1	On a write: Set Claim Tag bit <m> to 1.

Access to this field is RAO/W1S.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing DBGCLAIMSET_EL1

DBGCLAIMSET_EL1 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xFA0	DBGCLAIMSET_EL1

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), accesses to this register return an ERROR.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.2.6 DBGDTRRX_EL0, Debug Data Transfer Register, Receive

The DBGDTRRX_EL0 characteristics are:

Purpose

Transfers data from an external debugger to the PE. For example, it is used by a debugger transferring commands and data to a debug target. See DBGDTR_EL0 for additional architectural mappings. It is a component of the Debug Communications Channel.

Configuration

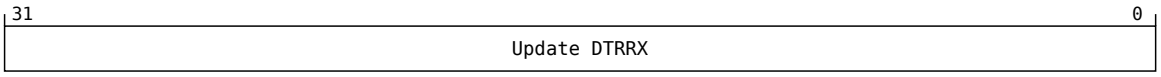
DBGDTRRX_EL0 is in the Core power domain

External register DBGDTRRX_EL0 bits [31:0] are architecturally mapped to AArch64 System register DBGDTRRX_EL0[31:0].

Attributes

DBGDTRRX_EL0 is a 32-bit register.

Field descriptions



DTRRX, bits [31:0]

Update DTRRX.

Writes to this register:

- If RXfull is 0, update the value in DTRRX and set RXfull to 1.
- If RXfull is 1, the written value is ignored and RXO is set to 1.

Reads of this register return the last value written to DTRRX and do not change RXfull.

For the full behavior of the Debug Communications Channel, see 'The Debug Communication Channel and Instruction Transfer Register'.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing DBGDTRRX_EL0

If EDSCR.ITE == 0 when the PE exits Debug state on receiving a Restart request trigger event, the behavior of any operation issued by a DTR access in Memory access mode that has not completed execution is CONSTRAINED UNPREDICTABLE, and must do one of the following:

- It must complete execution in Debug state before the PE executes the restart sequence.
- It must complete execution in Non-debug state before the PE executes the restart sequence.
- It must be abandoned. This means that the instruction does not execute. Any registers or memory accessed by the instruction are left in an UNKNOWN state.

DBGDTRRX_EL0 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0x080	DBGDTRRX_EL0

Accessible as follows:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), accesses to this register return an ERROR.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.2.7 DBGDTRTX_EL0, Debug Data Transfer Register, Transmit

The DBGDTRTX_EL0 characteristics are:

Purpose

Transfers data from the PE to an external debugger. For example, it is used by a debug target to transfer data to the debugger. See DBGDTR_EL0 for additional architectural mappings. It is a component of the Debug Communication Channel.

Configuration

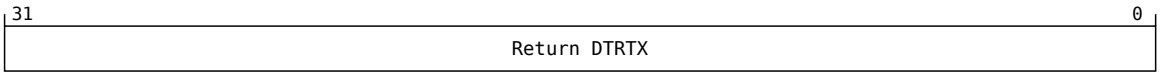
DBGDTRTX_EL0 is in the Core power domain

External register DBGDTRTX_EL0 bits [31:0] are architecturally mapped to AArch64 System register DBGDTRTX_EL0[31:0].

Attributes

DBGDTRTX_EL0 is a 32-bit register.

Field descriptions



DTRTX, bits [31:0]

Return DTRTX.

Reads of this register:

- If TXfull is 1, return the value in DTRTX and clear TXfull to 0.
- If TXfull is 0, return an UNKNOWN value and set TXU to 1.

Writes of this register update the value in DTRTX and do not change TXfull.

For the full behavior of the Debug Communications Channel, see 'The Debug Communication Channel and Instruction Transfer Register'.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing DBGDTRTX_EL0

If EDSCR.ITE == 0 when the PE exits Debug state on receiving a Restart request trigger event, the behavior of any operation issued by a DTR access in Memory access mode that has not completed execution is CONSTRAINED UNPREDICTABLE, and must do one of the following:

- It must complete execution in Debug state before the PE executes the restart sequence.
- It must complete execution in Non-debug state before the PE executes the restart sequence.
- It must be abandoned. This means that the instruction does not execute. Any registers or memory accessed by the instruction are left in an UNKNOWN state.

DBGDTRTX_EL0 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0x08C	DBGDTRTX_EL0

Accessible as follows:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), accesses to this register return an ERROR.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.2.8 DBGWCR<n>_EL1, Debug Watchpoint Control Registers, n = 0 - 63

The DBGWCR<n>_EL1 characteristics are:

Purpose

Holds control information for a watchpoint. Forms watchpoint n together with value register DBGWVR<n>_EL1.

Configuration

DBGWCR<n>_EL1 is in the Core power domain

If watchpoint n is not implemented, then accesses to this register are:

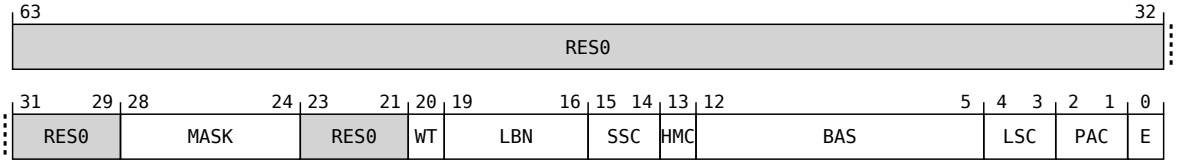
- When IsCorePowered() && !DoubleLockStatus() && !OSLockStatus() && AllowExternalDebugAccess(), RES0.
- Otherwise, a CONSTRAINED UNPREDICTABLE choice of RES0 or ERROR.

External register DBGWCR<n>_EL1 bits [31:0] are architecturally mapped to AArch64 System register DBGWCR<n>_EL1[31:0].

Attributes

DBGWCR<n>_EL1 is a 64-bit register.

Field descriptions



Bits [63:29]

Reserved, RES0.

MASK, bits [28:24]

Address Mask. Only address ranges up to 2GB can be watched using a single mask.

MASK	Meaning
0b00000	No mask.
0b00011..0b11111	Number of address bits masked.

All other values are reserved.

Indicates the number of masked address bits, from 0b00011 masking 3 address bits (0x00000007 mask for address) to 0b11111 masking 31 address bits (0x7FFFFFFF mask for address).

If programmed with a reserved value, the watchpoint behaves as if one of the following:

- DBGWCR<n>_EL1.MASK has been programmed with a defined value, which might be 0b00000 (no mask), other than for a direct read of DBGWCR<n>_EL1.
- The watchpoint is disabled.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [23:21]

Reserved, RES0.

WT, bit [20]

Watchpoint type. Possible values are:

WT	Meaning
0b0	Unlinked data address match.
0b1	Linked data address match.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

LBN, bits [19:16]

Linked Breakpoint Number.

For Linked data address watchpoints, specifies the index of the breakpoint linked to.

For all other watchpoint types, this field is ignored and reads of the register return an UNKNOWN value.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

SSC, bits [15:14]

Security state control. Determines the Security states under which a Watchpoint debug event for watchpoint n is generated. This field must be interpreted along with the HMC and PAC fields.

For more information on the operation of the SSC, HMC, and PAC fields, see 'Execution conditions for which a watchpoint generates Watchpoint exceptions'.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

HMC, bit [13]

Higher mode control. Determines the debug perspective for deciding when a Watchpoint debug event for watchpoint n is generated. This field must be interpreted along with the SSC and PAC fields.

For more information on the operation of the SSC, HMC, and PAC fields, see 'Execution conditions for which a watchpoint generates Watchpoint exceptions'.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

BAS, bits [12:5]

Byte address select. Each bit of this field selects whether a byte from within the word or double-word addressed by DBGWVR<n>_EL1 is being watched.

BAS	Description
xxxxxxx1	Match byte at DBGWVR<n>_EL1
xxxxxx1x	Match byte at DBGWVR<n>_EL1 + 1
xxxxx1xx	Match byte at DBGWVR<n>_EL1 + 2

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

BAS	Description
xxxx1xxx	Match byte at DBGWVR<n>_EL1 + 3

In cases where DBGWVR<n>_EL1 addresses a double-word:

BAS	Description, if DBGWVR<n>_EL1[2] == 0
xxx1xxxx	Match byte at DBGWVR<n>_EL1 + 4
xx1xxxxx	Match byte at DBGWVR<n>_EL1 + 5
x1xxxxxx	Match byte at DBGWVR<n>_EL1 + 6
1xxxxxxx	Match byte at DBGWVR<n>_EL1 + 7

If DBGWVR<n>_EL1[2] == 1, only BAS[3:0] is used. Arm deprecates setting DBGWVR<n>_EL1[2] == 1.

The valid values for BAS are nonzero binary number all of whose set bits are contiguous. All other values are reserved and must not be used by software. See 'Reserved DBGWCR<n>.BAS values'.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

LSC, bits [4:3]

Load/store control. This field enables watchpoint matching on the type of access being made. Possible values of this field are:

LSC	Meaning
0b01	Match instructions that load from a watchpointed address.
0b10	Match instructions that store to a watchpointed address.
0b11	Match instructions that load from or store to a watchpointed address.

All other values are reserved, but must behave as if the watchpoint is disabled. Software must not rely on this property as the behavior of reserved values might change in a future revision of the architecture.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

PAC, bits [2:1]

Privilege of access control. Determines the Exception level or levels at which a Watchpoint debug event for watchpoint n is generated. This field must be interpreted along with the SSC and HMC fields.

For more information on the operation of the SSC, HMC, and PAC fields, see 'Execution conditions for which a watchpoint generates Watchpoint exceptions'.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

E, bit [0]

Enable watchpoint n.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E	Meaning
0b0	Watchpoint n disabled.
0b1	Watchpoint n enabled.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing DBGWCR<n>_EL1

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalDebugAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

This register is 32-bits wide and offset 0x80C + (16 * n) is reserved.

DBGWCR<n>_EL1 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0x808 + (16 * n)	DBGWCR<n>_EL1

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), or !AllowExternalDebugAccess(addrdesc), accesses to this register return an ERROR.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.2.9 DBGWVR<n>_EL1, Debug Watchpoint Value Registers, n = 0 - 63

The DBGWVR<n>_EL1 characteristics are:

Purpose

Holds a data address value for use in watchpoint matching. Forms watchpoint n together with control register DBGWCR<n>_EL1.

Configuration

DBGWVR<n>_EL1 is in the Core power domain

If watchpoint n is not implemented, then accesses to this register are:

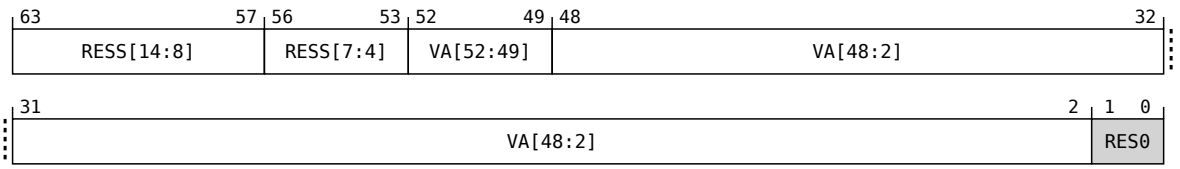
- When IsCorePowered() && !DoubleLockStatus() && !OSLockStatus() && AllowExternalDebugAccess(), RES0.
- Otherwise, a CONSTRAINED UNPREDICTABLE choice of RES0 or ERROR.

External register DBGWVR<n>_EL1 bits [63:0] are architecturally mapped to AArch64 System register DBGWVR<n>_EL1[63:0].

Attributes

DBGWVR<n>_EL1 is a 64-bit register.

Field descriptions



RESS[14:8], bits [63:57]

Reserved, Sign extended. Hardware and software must treat this field as RES0 if the most significant bit of VA is 0 or RES0, and as RES1 if the most significant bit of VA is 1.

Hardware always ignores the value of these bits and it is IMPLEMENTATION DEFINED whether:

- The bits are hardwired to a copy of the most significant bit of VA, meaning writes to these bits are ignored, and reads to the bits always return the hardwired value.
- The value in those bits can be written, and reads will return the last value written. The value held in those bits is ignored by hardware.

RESS[7:4], bits [56:53]

Extension to RESS[14:8]. For more information, see RESS[14:8].

Bits [52:49]

When FEAT_LVA is implemented:

VA[52:49], bits [52:49]

Extension to VA[48:2]. For more information, see VA[48:2].

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RESS[3:0], bits [52:49]

Extension to RESS[14:8]. For more information, see RESS[14:8].

VA[48:2], bits [48:2]

Bits[48:2] of the address value for comparison.

When FEAT_LVA is implemented, VA[52:49] forms the upper part of the address value. If FEAT_LVA is not implemented, bits [52:49] are part of the RESS field.

Arm deprecates setting DBGWVR<n>_EL1[2] == 1.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [1:0]

Reserved, RES0.

Accessing DBGWVR<n>_EL1

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalDebugAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

DBGWVR<n>_EL1 can be accessed through the external debug interface:

Component	Offset	Instance	Range
Debug	`0x800 + (16 * n)`	DBGWVR<n>_EL1	63:0

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), or !AllowExternalDebugAccess(addrdesc), accesses to this register return an ERROR.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.2.10 EDAA32PFR, External Debug Auxiliary Processor Feature Register

The EDAA32PFR characteristics are:

Purpose

Provides information about implemented PE features.

Note

The register mnemonic, EDAA32PFR, is derived from previous definitions of this register that defined this register only when AArch64 was not supported.

For general information about the interpretation of the ID registers, see 'Principles of the ID scheme for fields in ID registers'.

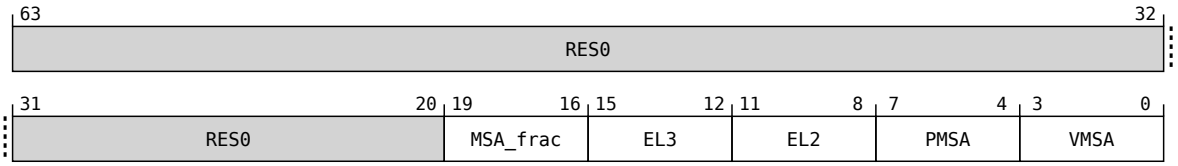
Configuration

There are no configuration notes.

Attributes

EDAA32PFR is a 64-bit register.

Field descriptions



Bits [63:20]

Reserved, RES0.

MSA_frac, bits [19:16]

When EDAA32PFR.PMSA == '0000' and EDAA32PFR.VMSA == '1111':

Memory System Architecture fractional field. This holds the information on additional Memory System Architectures supported.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MSA_frac	Meaning
0b0001	PMSAv8-64 supported in all translation regimes. VMSAv8-64 not supported.
0b0010	PMSAv8-64 supported in all translation regimes. In addition to PMSAv8-64, stage 1 EL1&0 translation regime also supports VMSAv8-64.

All other values are reserved.

In Armv8-R AArch64, the only permitted values are 0b0001 and 0b0010.

Access to this field is RO.

Otherwise:

Reserved, RES0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EL3, bits [15:12]

When EDPFR.EL3 == '0000':

AArch32 EL3 Exception level handling.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EL3	Meaning
0b0000	EL3 is not implemented or can be executed in AArch64 state.
0b0001	EL3 can be executed in AArch32 state only.

All other values are reserved.

In an Armv8-R AArch64 implementation this field returns the value of 0b0000.

Access to this field is RO.

Otherwise:

Reserved, RAZ.

EL2, bits [11:8]

When EDPFR.EL2 == '0000':

AArch32 EL2 Exception level handling.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EL2	Meaning
0b0000	EL2 is not implemented or can be executed in AArch64 state.
0b0001	EL2 can be executed in AArch32 state only.

All other values are reserved.

In an Armv8-R AArch64 implementation this field returns the value of 0b0000.

Access to this field is RO.

Otherwise:

Reserved, RAZ.

PMSA, bits [7:4]

Indicates support for a 32-bit PMSA.

The value of this field is an IMPLEMENTATION DEFINED choice of:

PMSA	Meaning
0b0000	PMSA-32 not supported.
0b0100	PMSAv8-32 supported.

Access to this field is RO.

VMSA, bits [3:0]

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When EDAA32PFR.PMSA != '0000':

Indicates support for a VMSA in addition to a 32-bit PMSA.

VMSA	Meaning
0b0000	VMSA not supported.

All other values are reserved.

Access to this field is RO.

When EDAA32PFR.PMSA == '0000':

The value of this field is an IMPLEMENTATION DEFINED choice of:

VMSA	Meaning
0b0000	VMSAv8-64 supported. PMSAv8-64 not supported.
0b1111	Memory system architecture described by EDAA32PFR.MSA_frac.

All other values are reserved.

In Armv8-R AArch64, the only permitted value is 0b1111.

Access to this field is RO.

Otherwise:

Reserved, RAZ.

Accessing EDAA32PFR

EDAA32PFR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xD60	EDAA32PFR

Accessible as follows:

- When IsCorePowered() and !DoubleLockStatus(), accesses to this register are RO.
- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are IMPLEMENTATION DEFINED.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.2.11 EDACR, External Debug Auxiliary Control Register

The EDACR characteristics are:

Purpose

Allows implementations to support IMPLEMENTATION DEFINED controls.

Configuration

When FEAT_DoPD is implemented, EDACR is in the Core power domain. Otherwise, it is IMPLEMENTATION DEFINED whether EDACR is implemented in the Core power domain or in the Debug power domain

If this register is implemented, EDDEVID.AuxRegs == `0b0001`.

If FEAT_DoPD is implemented, any mechanism to preserve control bits in EDACR over power down is optional and IMPLEMENTATION DEFINED.

If FEAT_DoPD is not implemented and EDACR contains any control bits that must be preserved over power down, then these bits must be accessible by the external debug interface when the OS Lock is locked, OSLSR_EL1.OSLK == 1, and when the Core is powered off.

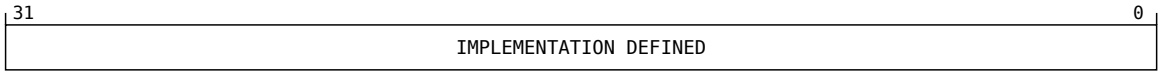
Changing this register from its reset value causes IMPLEMENTATION DEFINED behavior, including possible deviation from the architecturally-defined behavior.

Implementation of this register is OPTIONAL.

Attributes

EDACR is a 32-bit register.

Field descriptions



IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

The following resets apply:

- If the register is implemented in the Core power domain:
  - On a Cold reset, this field resets to an architecturally UNKNOWN value.
  - On an External debug reset, the value of this field is unchanged.
  - On a Warm reset, the value of this field is unchanged.
- If the register is implemented in the External debug power domain:
  - On a Cold reset, the value of this field is unchanged.
  - On an External debug reset, this field resets to an architecturally UNKNOWN value.
  - On a Warm reset, the value of this field is unchanged.

Accessing EDACR

EDACR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0x094`	EDACR

Accessible as follows:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), accesses to this register are IMPLEMENTATION DEFINED.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.2.12 EDCIDR0, External Debug Component Identification Register 0

The EDCIDR0 characteristics are:

Purpose

Provides information to identify an external debug component.

For more information, see ‘About the Component Identification scheme’.

Configuration

When FEAT_DoPD is implemented, EDCIDR0 is in the Core power domain. Otherwise, EDCIDR0 is in the Debug power domain

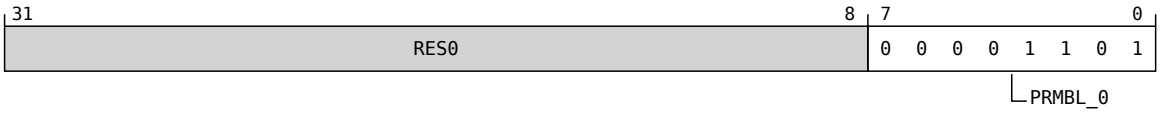
This register is required for CoreSight compliance.

Implementation of this register is OPTIONAL.

Attributes

EDCIDR0 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

PRMBL_0, bits [7:0]

Preamble.

Reads as `0x0D`

Access to this field is RO.

Accessing EDCIDR0

EDCIDR0 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0xFF0`	EDCIDR0

Accessible as follows:

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.2.13 EDCIDR1, External Debug Component Identification Register 1

The EDCIDR1 characteristics are:

Purpose

Provides information to identify an external debug component.

For more information, see ‘About the Component Identification scheme’.

Configuration

When FEAT_DoPD is implemented, EDCIDR1 is in the Core power domain. Otherwise, EDCIDR1 is in the Debug power domain

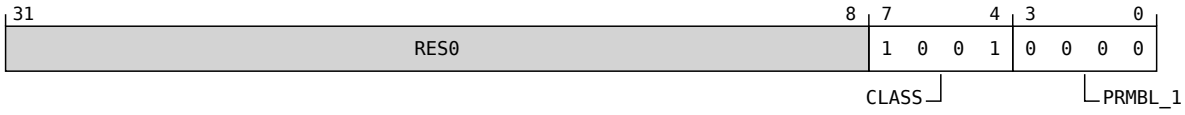
This register is required for CoreSight compliance.

Implementation of this register is OPTIONAL.

Attributes

EDCIDR1 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

CLASS, bits [7:4]

Component class.

CLASS	Meaning
0b1001	CoreSight component.

Other values are defined by the CoreSight Architecture.

This field reads as 0x9.

Access to this field is RO.

PRMBL_1, bits [3:0]

Preamble.

Reads as 0b0000

Access to this field is RO.

Accessing EDCIDR1

EDCIDR1 can be accessed through the external debug interface:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Offset	Instance
Debug	0xFF4	EDCIDR1

Accessible as follows:

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.2.14 EDCIDR2, External Debug Component Identification Register 2

The EDCIDR2 characteristics are:

Purpose

Provides information to identify an external debug component.

For more information, see ‘About the Component Identification scheme’.

Configuration

When FEAT_DoPD is implemented, EDCIDR2 is in the Core power domain. Otherwise, EDCIDR2 is in the Debug power domain

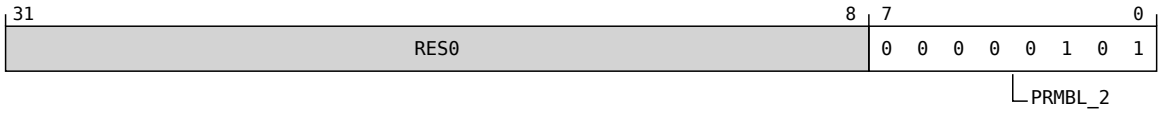
This register is required for CoreSight compliance.

Implementation of this register is OPTIONAL.

Attributes

EDCIDR2 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

PRMBL_2, bits [7:0]

Preamble.

Reads as `0x05`

Access to this field is RO.

Accessing EDCIDR2

EDCIDR2 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0xFF8`	EDCIDR2

Accessible as follows:

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.2.15 EDCIDR3, External Debug Component Identification Register 3

The EDCIDR3 characteristics are:

Purpose

Provides information to identify an external debug component.

For more information, see ‘About the Component Identification scheme’.

Configuration

When FEAT_DoPD is implemented, EDCIDR3 is in the Core power domain. Otherwise, EDCIDR3 is in the Debug power domain

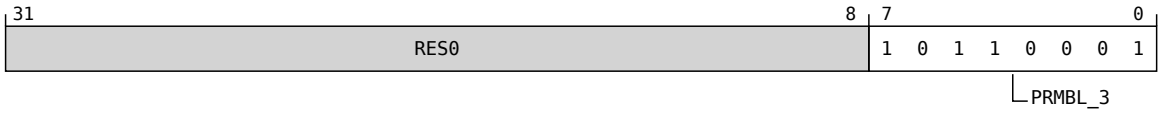
This register is required for CoreSight compliance.

Implementation of this register is OPTIONAL.

Attributes

EDCIDR3 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

PRMBL_3, bits [7:0]

Preamble.

Reads as `0xB1`

Access to this field is RO.

Accessing EDCIDR3

EDCIDR3 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0xFFC`	EDCIDR3

Accessible as follows:

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.2.16 EDDEVAFF0, External Debug Device Affinity register 0

The EDDEVAFF0 characteristics are:

Purpose

Copy of the low half of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the external debug component relates to.

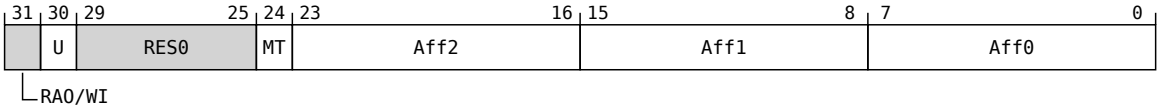
Configuration

When FEAT_DoPD is implemented, EDDEVAFF0 is in the Core power domain. Otherwise, EDDEVAFF0 is in the Debug power domain

Attributes

EDDEVAFF0 is a 32-bit register.

Field descriptions



Bit [31]

Reserved, RAO/WI.

U, bit [30]

Indicates a Uniprocessor system, as distinct from PE 0 in a multiprocessor system.

The value of this field is an IMPLEMENTATION DEFINED choice of:

U	Meaning
0b0	Processor is part of a multiprocessor system.
0b1	Processor is part of a uniprocessor system.

Access to this field is RO.

Bits [29:25]

Reserved, RES0.

MT, bit [24]

Indicates whether the lowest level of affinity consists of logical PEs that are implemented using an interdependent approach, such as multithreading. See the description of Aff0 for more information about affinity levels.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MT	Meaning
0b0	Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is largely independent.
0b1	Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is very interdependent.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

This field does not indicate that multithreading is implemented and does not indicate that PEs with different affinity level 0 values, and the same values for affinity level 1 and higher are implemented.

Access to this field is RO.

Aff2, bits [23:16]

Affinity level 2. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Aff1, bits [15:8]

Affinity level 1. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Aff0, bits [7:0]

Affinity level 0. The value of the MPIDR.{Aff2, Aff1, Aff0} or MPIDR_EL1.{Aff3, Aff2, Aff1, Aff0} set of fields of each PE must be unique within the system as a whole.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing EDDEVAFF0

EDDEVAFF0 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xFA8	EDDEVAFF0

Accessible as follows:

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.2.17 EDDEVAFF1, External Debug Device Affinity register 1

The EDDEVAFF1 characteristics are:

Purpose

Copy of the high half of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the external debug component relates to.

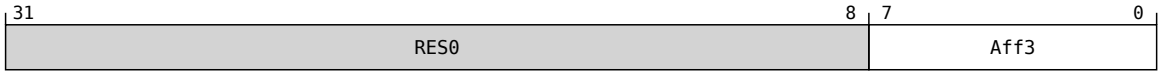
Configuration

When FEAT_DoPD is implemented, EDDEVAFF1 is in the Core power domain. Otherwise, EDDEVAFF1 is in the Debug power domain

Attributes

EDDEVAFF1 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

Aff3, bits [7:0]

Affinity level 3. See the description of EDDEVAFF0.Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing EDDEVAFF1

EDDEVAFF1 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xFAC	EDDEVAFF1

Accessible as follows:

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.2.18 EDDEVARCH, External Debug Device Architecture Register

The EDDEVARCH characteristics are:

Purpose

Identifies the programmers' model architecture of the external debug component.

Configuration

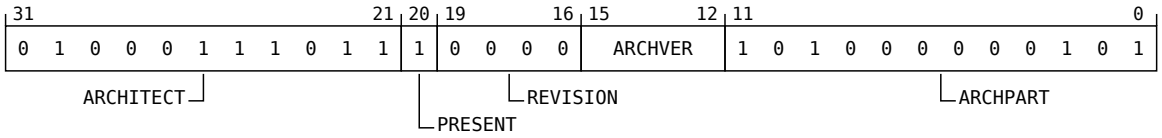
When FEAT_DoPD is implemented, EDDEVARCH is in the Core power domain. Otherwise, EDDEVARCH is in the Debug power domain

Implementation of this register is OPTIONAL.

Attributes

EDDEVARCH is a 32-bit register.

Field descriptions



ARCHITECT, bits [31:21]

Defines the architect of the component. For External Debug, this is Arm Limited.

Bits [31:28] are the JEP106 continuation code, `0b0100`.

Bits [27:21] are the JEP106 identification code, `0b0111011`.

Reads as `0b01000111011`

Access to this field is RO.

PRESENT, bit [20]

DEVARCH present. Indicates that the EDDEVARCH register is present.

Reads as `0b1`

Access to this field is RO.

REVISION, bits [19:16]

Defines the architecture revision. For architectures defined by Arm this is the minor revision.

For debug, the revision defined by Armv8 is `0x0`.

All other values are reserved.

Reads as `0b0000`

Access to this field is RO.

ARCHVER, bits [15:12]

Architecture Version. Defines the architecture version of the component.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EDDEVARCH.ARCHVER and EDDEVARCH.ARCHPART are also defined as a single field, EDDEVARCH.ARCHID, so that EDDEVARCH.ARCHVER is EDDEVARCH.ARCHID[15:12].

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_Debugv8p1 implements the functionality identified by the value 0b0111.

FEAT_Debugv8p2 implements the functionality identified by the value 0b1000.

FEAT_Debugv8p4 implements the functionality identified by the value 0b1001.

From Armv8.1, when FEAT_Debugv8p1 is implemented the value 0b0110 is not permitted.

From Armv8.2, the values 0b0110 and 0b0111 are not permitted.

From Armv8.4, the value 0b1000 is not permitted.

Access to this field is RO.

ARCHPART, bits [11:0]

Architecture Part. Defines the architecture of the component.

ARCHPART	Meaning
0xA05	Armv8-R AArch64 debug architecture.

EDDEVARCH.ARCHVER and EDDEVARCH.ARCHPART are also defined as a single field, EDDEVARCH.ARCHID, so that EDDEVARCH.ARCHPART is EDDEVARCH.ARCHID[11:0].

Armv8-R debug architecture.

Access to this field is RO.

Accessing EDDEVARCH

EDDEVARCH can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xFBC	EDDEVARCH

Accessible as follows:

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.2.19 EDDEVID, External Debug Device ID register 0

The EDDEVID characteristics are:

Purpose

Provides extra information for external debuggers about features of the debug implementation.

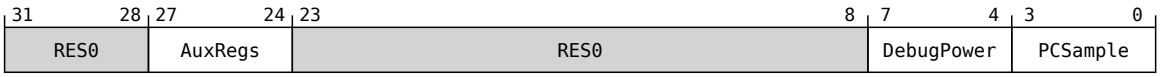
Configuration

When FEAT_DoPD is implemented, EDDEVID is in the Core power domain. Otherwise, EDDEVID is in the Debug power domain

Attributes

EDDEVID is a 32-bit register.

Field descriptions



Bits [31:28]

Reserved, RES0.

AuxRegs, bits [27:24]

Indicates support for Auxiliary registers.

The value of this field is an IMPLEMENTATION DEFINED choice of:

AuxRegs	Meaning
0b0000	None supported.
0b0001	Support for External Debug Auxiliary Control Register, EDACR.

All other values are reserved.

Access to this field is RO.

Bits [23:8]

Reserved, RES0.

DebugPower, bits [7:4]

Indicates support for the FEAT_DoPD feature.

The value of this field is an IMPLEMENTATION DEFINED choice of:

DebugPower	Meaning
0b0000	FEAT_DoPD not implemented. Registers in the external debug interface register map are implemented in a mix of the Debug and Core power domains.
0b0001	FEAT_DoPD implemented. All registers in the external debug interface register map are implemented in the Core power domain.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FEAT_DoPD implements the functionality added by the value 0b0001.

All other values are reserved.

Access to this field is RO.

PCSample, bits [3:0]

Indicates the level of PC Sample-based Profiling support using external debug registers.

The value of this field is an IMPLEMENTATION DEFINED choice of:

PCSample	Meaning
0b0000	PC Sample-based Profiling Extension is not implemented in the external debug registers space.
0b0010	Only EDPCSR and EDCIDSR are implemented. This option is only permitted if EL3 and EL2 are not implemented.
0b0011	EDPCSR, EDCIDSR, and EDVIDSR are implemented.

All other values are reserved.

When FEAT_PCSRv8p2 is implemented, the only permitted value is 0b0000.

Note

FEAT_PCSRv8p2 implements the PC Sample-based Profiling Extension in the Performance Monitors register space, as indicated by the value of PMDEVID.PCSample.

Access to this field is RO.

Accessing EDDEVID

EDDEVID can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xFC8	EDDEVID

Accessible as follows:

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.2.20 EDDEVID1, External Debug Device ID Register 1

The EDDEVID1 characteristics are:

Purpose

Provides extra information for external debuggers about features of the debug implementation.

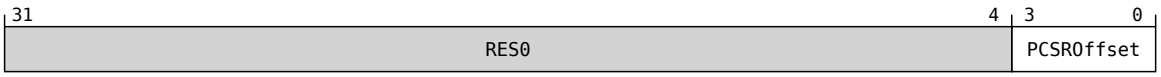
Configuration

When FEAT_DoPD is implemented, EDDEVID1 is in the Core power domain. Otherwise, EDDEVID1 is in the Debug power domain

Attributes

EDDEVID1 is a 32-bit register.

Field descriptions



Bits [31:4]

Reserved, RES0.

PCSROffset, bits [3:0]

Indicates the offset applied to PC samples returned by reads of EDPCSR.

The value of this field is an IMPLEMENTATION DEFINED choice of:

PCSROffset	Meaning
0b0000	EDPCSR not implemented.
0b0010	EDPCSR implemented, and samples have no offset applied.

When FEAT_PCSRv8p2 is implemented, the only permitted value is 0b0000.

Note

FEAT_PCSRv8p2 implements the PC Sample-based Profiling Extension in the Performance Monitors register space, as indicated by the value of PMDEVID.PCSample.

Access to this field is RO.

Accessing EDDEVID1

EDDEVID1 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xFC4	EDDEVID1

Accessible as follows:

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.2.21 EDDEVID2, External Debug Device ID register 2

The EDDEVID2 characteristics are:

Purpose

Reserved for future descriptions of features of the debug implementation.

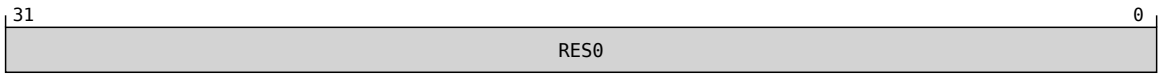
Configuration

When FEAT_DoPD is implemented, EDDEVID2 is in the Core power domain. Otherwise, EDDEVID2 is in the Debug power domain

Attributes

EDDEVID2 is a 32-bit register.

Field descriptions



Bits [31:0]

Reserved, RES0.

Accessing EDDEVID2

EDDEVID2 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xFC0	EDDEVID2

Accessible as follows:

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.2.22 EDDEVTYPE, External Debug Device Type register

The EDDEVTYPE characteristics are:

Purpose

Indicates to a debugger that this component is part of a PE's debug logic.

Configuration

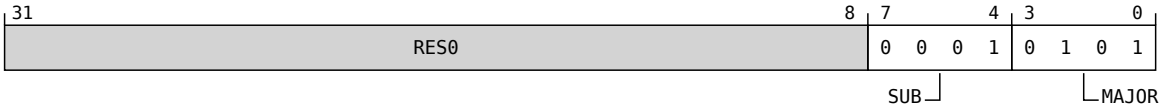
When FEAT_DoPD is implemented, EDDEVTYPE is in the Core power domain. Otherwise, EDDEVTYPE is in the Debug power domain

Implementation of this register is OPTIONAL.

Attributes

EDDEVTYPE is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

SUB, bits [7:4]

Subtype. Indicates this is a component within a PE.

Reads as `0b0001`

Access to this field is RO.

MAJOR, bits [3:0]

Major type. Indicates this is a debug logic component.

Reads as `0b0101`

Access to this field is RO.

Accessing EDDEVTYPE

EDDEVTYPE can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0xFCC`	EDDEVTYPE

Accessible as follows:

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.2.23 EDDFR, External Debug Feature Register

The EDDFR characteristics are:

Purpose

Provides top-level information about the debug system.

Note

Debuggers must use EDDEVARCH to determine the Debug architecture version.

For general information about the interpretation of the ID registers, see 'Principles of the ID scheme for fields in ID registers'.

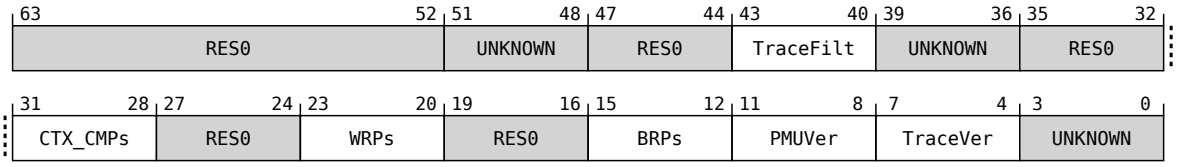
Configuration

There are no configuration notes.

Attributes

EDDFR is a 64-bit register.

Field descriptions



Bits [63:52]

Reserved, RES0.

Bits [51:48]

Reserved, UNKNOWN.

Bits [47:44]

Reserved, RES0.

TraceFilt, bits [43:40]

Armv8.4 Self-hosted Trace Extension version.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TraceFilt	Meaning
0b0000	Armv8.4 Self-hosted Trace Extension not implemented.
0b0001	Armv8.4 Self-hosted Trace Extension implemented.

All other values are reserved.

FEAT_TRF implements the functionality identified by the value 0b0001.

From Armv8.4, if FEAT_ETMv4 is implemented, the value 0b0000 is not permitted.

Access to this field is RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [39:36]

Reserved, UNKNOWN.

Bits [35:32]

Reserved, RES0.

CTX_CMPs, bits [31:28]

Number of context-aware breakpoints, minus 1.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CTX_CMPs	Meaning
0b0000..0b1111	The number of context-aware breakpoints, minus 1.

The value of this field is never greater than EDDFR.BRPs.

This field has the same value as ID_AA64DFR0_EL1.CTX_CMPs.

Access to this field is RO.

Bits [27:24]

Reserved, RES0.

WRPs, bits [23:20]

Number of watchpoints, minus 1.

The value of this field is an IMPLEMENTATION DEFINED choice of:

WRPs	Meaning
0b0001..0b1111	The number of watchpoints, minus 1.

This field has the same value as ID_AA64DFR0_EL1.WRPs.

The value 0b0000 is reserved.

Access to this field is RO.

Bits [19:16]

Reserved, RES0.

BRPs, bits [15:12]

Number of breakpoints, minus 1.

The value of this field is an IMPLEMENTATION DEFINED choice of:

BRPs	Meaning
0b0001..0b1111	The number of breakpoints, minus 1.

This field has the same value as ID_AA64DFR0_EL1.BRPs.

The value 0b0000 is reserved.

Access to this field is RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PMUVer, bits [11:8]

Performance Monitors Extension version.

This field does not follow the standard ID scheme, but uses the alternative ID scheme described in 'Alternative ID scheme used for the Performance Monitors Extension version'

The value of this field is an IMPLEMENTATION DEFINED choice of:

All other values are reserved.

FEAT_PMUv3 implements the functionality identified by the value `0b0001`.

FEAT_PMUv3p1 implements the functionality identified by the value `0b0100`.

FEAT_PMUv3p4 implements the functionality identified by the value `0b0101`.

From Armv8.1, if FEAT_PMUv3 is implemented, the value `0b0001` is not permitted.

From Armv8.4, if FEAT_PMUv3 is implemented, the value `0b0100` is not permitted.

This field has the same value as ID_AA64DFR0_EL1.PMUVer.

Access to this field is RO.

TraceVer, bits [7:4]

Trace support. Indicates whether System register interface to a trace unit is implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TraceVer	Meaning
`0b0000`	Trace unit System registers not implemented.
`0b0001`	Trace unit System registers implemented.

All other values are reserved.

A value of `0b0000` only indicates that no System register interface to a trace unit is implemented. A trace unit might nevertheless be implemented without a System register interface.

In an Armv8-A implementation that supports AArch64, this field returns the value of ID_AA64DFR0_EL1.TraceVer.

Access to this field is RO.

Bits [3:0]

Reserved, UNKNOWN.

Accessing EDDFR

EDDFR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0xD28`	EDDFR

Accessible as follows:

- When IsCorePowered() and !DoubleLockStatus(), accesses to this register are RO.
- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are IMPLEMENTATION DEFINED.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.2.24 EDECCR, External Debug Exception Catch Control Register

The EDECCR characteristics are:

Purpose

Controls Exception Catch debug events. For more information, see 'Exception Catch debug event'.

Configuration

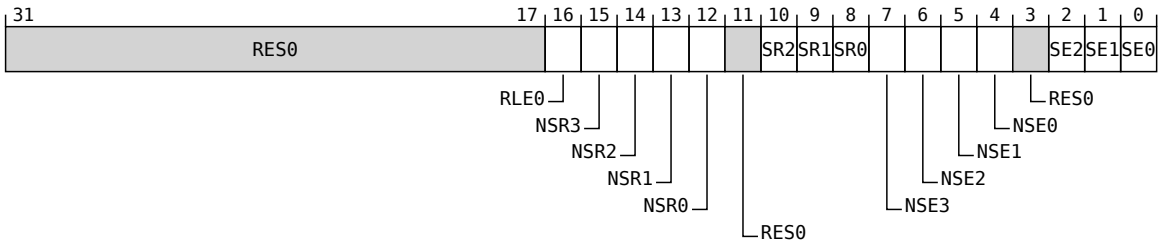
EDECCR is in the Core power domain

External register EDECCR bits [31:0] are architecturally mapped to AArch64 System register OSECCR_EL1[31:0].

Attributes

EDECCR is a 32-bit register.

Field descriptions



Bits [31:17]

Reserved, RES0.

RLE0, bit [16]

Access to this field is RES0.

NSR3, bit [15]

Access to this field is RES0.

NSR2, bit [14]

When FEAT_Debugv8p2 is implemented and Non-secure EL2 is implemented:

Controls exception catch on exception return to Non-secure EL2 in conjunction with EDECCR.NSE2.

NSR2	Meaning
0b0	If EDECCR.NSE2 is 0, then Exception Catch debug events are disabled for Non-secure EL2. If EDECCR.NSE2 is 1, then Exception Catch debug events are enabled for exception entry, reset entry, and exception return to Non-secure EL2.
0b1	If EDECCR.NSE2 is 0, then Exception Catch debug events are enabled for exception returns to Non-secure EL2. If EDECCR.NSE2 is 1, then Exception Catch debug events are enabled for exception entry and reset entry to Non-secure EL2.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

NSR1, bit [13]

When FEAT_Debugv8p2 is implemented and Non-secure EL1 is implemented:

Controls exception catch on exception return to Non-secure EL1 in conjunction with EDECCR.NSE1.

NSR1	Meaning
0b0	If EDECCR.NSE1 is 0, then Exception Catch debug events are disabled for Non-secure EL1. If EDECCR.NSE1 is 1, then Exception Catch debug events are enabled for exception entry, reset entry, and exception return to Non-secure EL1.
0b1	If EDECCR.NSE1 is 0, then Exception Catch debug events are enabled for exception returns to Non-secure EL1. If EDECCR.NSE1 is 1, then Exception Catch debug events are enabled for exception entry and reset entry to Non-secure EL1.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Otherwise:

Reserved, RES0.

NSR0, bit [12]

When FEAT_Debugv8p2 is implemented and Non-secure EL0 is implemented:

Controls exception catch on exception return to Non-secure EL0.

NSR0	Meaning
0b0	Exception Catch debug events are disabled for Non-secure EL0.
0b1	Exception Catch debug events are enabled for exception returns to Non-secure EL0.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Otherwise:

Reserved, RES0.

Bit [11]

Reserved, RES0.

SR2, bit [10]

When FEAT_Debugv8p2 is implemented and FEAT_SEL2 is implemented:

Controls exception catch on exception return to Secure EL2 in conjunction with EDECCR.SE2.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SR2	Meaning
0b0	If EDECCR.SE2 is 0, then Exception Catch debug events are disabled for Secure EL2. If EDECCR.SE2 is 1, then Exception Catch debug events are enabled for exception entry, reset entry, and exception return to Secure EL2.
0b1	If EDECCR.SE2 is 0, then Exception Catch debug events are enabled for exception returns to Secure EL2. If EDECCR.SE2 is 1, then Exception Catch debug events are enabled for exception entry and reset entry to Secure EL2.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Otherwise:

Reserved, RES0.

SR1, bit [9]

When FEAT_Debugv8p2 is implemented and Secure EL1 is implemented:

Controls exception catch on exception return to Secure EL1 in conjunction with EDECCR.SE1.

SR1	Meaning
0b0	If EDECCR.SE1 is 0, then Exception Catch debug events are disabled for Secure EL1. If EDECCR.SE1 is 1, then Exception Catch debug events are enabled for exception entry, reset entry, and exception return to Secure EL1.
0b1	If EDECCR.SE1 is 0, then Exception Catch debug events are enabled for exception returns to Secure EL1. If EDECCR.SE1 is 1, then Exception Catch debug events are enabled for exception entry and reset entry to Secure EL1.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Otherwise:

Reserved, RES0.

SR0, bit [8]

When FEAT_Debugv8p2 is implemented and Secure EL0 is implemented:

Controls exception catch on exception return to Secure EL0.

SR0	Meaning
0b0	Exception Catch debug events are disabled for Secure EL0.
0b1	Exception Catch debug events are enabled for exception returns to Secure EL0.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Otherwise:

Reserved, RES0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NSE3, bit [7]

Access to this field is RES0.

NSE2, bit [6]

When FEAT_Debugv8p2 is implemented and Non-secure EL2 is implemented:

Controls exception catch on exception entry to Non-secure EL2. Also controls exception catch on exception return to Non-secure EL2 in conjunction with EDECCR.NSR2.

NSE2	Meaning
0b0	If EDECCR.NSR2 is 0, then Exception Catch debug events are disabled for Non-secure EL2. If EDECCR.NSR2 is 1, then Exception Catch debug events are enabled for exception returns to Non-secure EL2.
0b1	If EDECCR.NSR2 is 0, then Exception Catch debug events are enabled for exception entry, reset entry, and exception return to Non-secure EL2. If EDECCR.NSR2 is 1, then Exception Catch debug events are enabled for exception entry and reset entry to Non-secure EL2.

Note

It is IMPLEMENTATION DEFINED whether a reset entry to an Exception level will generate an Exception Catch debug event.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

When Non-secure EL2 is implemented:

Coarse-grained exception catch for Non-secure EL2. Controls Exception Catch debug events for Non-secure EL2.

NSE2	Meaning
0b0	Exception Catch debug events are disabled for Non-secure EL2.
0b1	Exception Catch debug events are enabled for Non-secure EL2.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Otherwise:

Reserved, RES0.

NSE1, bit [5]

When FEAT_Debugv8p2 is implemented and Non-secure EL1 is implemented:

Controls exception catch on exception entry to Non-secure EL1. Also controls exception catch on exception return to Non-secure EL1 in conjunction with EDECCR.NSR1.

NSE1	Meaning
0b0	If EDECCR.NSR1 is 0, then Exception Catch debug events are disabled for Non-secure EL1. If EDECCR.NSR1 is 1, then Exception Catch debug events are enabled for exception returns to Non-secure EL1.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NSE1	Meaning
0b1	If EDECCR.NSR1 is 0, then Exception Catch debug events are enabled for exception entry, reset entry, and exception return to Non-secure EL1. If EDECCR.NSR1 is 1, then Exception Catch debug events are enabled for exception entry and reset entry to Non-secure EL1.

Note

It is IMPLEMENTATION DEFINED whether a reset entry to an Exception level will generate an Exception Catch debug event.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

When Non-secure EL1 is implemented:

Coarse-grained exception catch for Non-secure EL1. Controls Exception Catch debug events for Non-secure EL1.

NSE1	Meaning
0b0	Exception Catch debug events are disabled for Non-secure EL1.
0b1	Exception Catch debug events are enabled for Non-secure EL1.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Otherwise:

Reserved, RES0.

NSE0, bit [4]

Access to this field is RES0.

Bit [3]

Reserved, RES0.

SE2, bit [2]

When FEAT_Debugv8p2 is implemented and FEAT_SEL2 is implemented:

Controls exception catch on exception entry to Secure EL2. Also controls exception catch on exception return to Secure EL2 in conjunction with EDECCR.SR2.

SE2	Meaning
0b0	If EDECCR.SR2 is 0, then Exception Catch debug events are disabled for Secure EL2. If EDECCR.SR2 is 1, then Exception Catch debug events are enabled for exception returns to Secure EL2.
0b1	If EDECCR.SR2 is 0, then Exception Catch debug events are enabled for exception entry, reset entry, and exception return to Secure EL2. If EDECCR.SR2 is 1, then Exception Catch debug events are enabled for exception entry and reset entry to Secure EL2.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

It is IMPLEMENTATION DEFINED whether a reset entry to an Exception level will generate an Exception Catch debug event.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Otherwise:

Reserved, RES0.

SE1, bit [1]

When FEAT_Debugv8p2 is implemented and Secure EL1 is implemented:

Controls exception catch on exception entry to Secure EL1. Also controls exception catch on exception return to Secure EL1 in conjunction with EDECCR.SR1.

SE1	Meaning
0b0	If EDECCR.SR1 is 0, then Exception Catch debug events are disabled for Secure EL1. If EDECCR.SR1 is 1, then Exception Catch debug events are enabled for exception returns to Secure EL1.
0b1	If EDECCR.SR1 is 0, then Exception Catch debug events are enabled for exception entry, reset entry, and exception return to Secure EL1. If EDECCR.SR1 is 1, then Exception Catch debug events are enabled for exception entry and reset entry to Secure EL1.

Note

It is IMPLEMENTATION DEFINED whether a reset entry to an Exception level will generate an Exception Catch debug event.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

When Secure EL1 is implemented:

Coarse-grained exception catch for Secure EL1. Controls Exception Catch debug events for Secure EL1.

SE1	Meaning
0b0	Exception Catch debug events are disabled for Secure EL1.
0b1	Exception Catch debug events are enabled for Secure EL1.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Otherwise:

Reserved, RES0.

SE0, bit [0]

Access to this field is RES0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing EDECCR

EDECCR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0x098	EDECCR

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), accesses to this register return an ERROR.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.2.25 EDECR, External Debug Execution Control Register

The EDECR characteristics are:

Purpose

Controls Halting debug events.

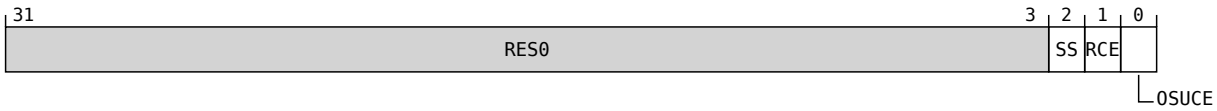
Configuration

When FEAT_DoPD is implemented, EDECR is in the Core power domain. Otherwise, EDECR is in the Debug power domain

Attributes

EDECR is a 32-bit register.

Field descriptions



Bits [31:3]

Reserved, RES0.

SS, bit [2]

Halting step enable. Possible values of this field are:

SS	Meaning
0b0	Halting step debug event disabled.
0b1	Halting step debug event enabled.

If the value of EDECR.SS is changed when the PE is in Non-debug state, behavior is CONSTRAINED UNPREDICTABLE as described in 'Changing the value of EDECR.SS when not in Debug state'.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_DoPD is implemented, this field resets to '0'.
- On a External debug reset:
  - When FEAT_DoPD is not implemented, this field resets to '0'.

RCE, bit [1]

When FEAT_DoPD is not implemented:

Reset Catch Enable.

RCE	Meaning
0b0	Reset Catch debug event disabled.
0b1	Reset Catch debug event enabled.

The reset behavior of this field is:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a External debug reset, this field resets to '0'.

Otherwise:

Reserved, RES0.

OSUCE, bit [0]

When FEAT_DoPD is not implemented:

OS Unlock Catch Enable.

OSUCE	Meaning
0b0	OS Unlock Catch debug event disabled.
0b1	OS Unlock Catch debug event enabled.

The reset behavior of this field is:

- On a External debug reset, this field resets to '0'.

Otherwise:

Reserved, RES0.

Accessing EDECR

EDECR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0x024	EDECR

Accessible as follows:

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.2.26 EDESR, External Debug Event Status Register

The EDESR characteristics are:

Purpose

Indicates the status of internally pending Halting debug events.

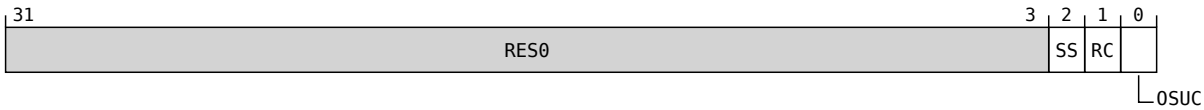
Configuration

EDESR is in the Core power domain

Attributes

EDESR is a 32-bit register.

Field descriptions



Bits [31:3]

Reserved, RES0.

SS, bit [2]

When FEAT_DoPD is implemented:

Halting step debug event pending. Possible values of this field are:

SS	Meaning
0b0	Reading this means that a Halting step debug event is not pending. Writing this means no action.
0b1	Reading this means that a Halting step debug event is pending. Writing this clears the pending Halting step debug event.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Otherwise:

Halting step debug event pending. Possible values of this field are:

SS	Meaning
0b0	Reading this means that a Halting step debug event is not pending. Writing this means no action.
0b1	Reading this means that a Halting step debug event is pending. Writing this clears the pending Halting step debug event.

The reset behavior of this field is:

- On a Warm reset, this field resets to the value in EDECR.SS.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RC, bit [1]

Reset Catch debug event pending. Possible values of this field are:

RC	Meaning
0b0	Reading this means that a Reset Catch debug event is not pending. Writing this means no action.
0b1	Reading this means that a Reset Catch debug event is pending. Writing this clears the pending Reset Catch debug event.

The reset behavior of this field is:

- On a Warm reset:
  - When FEAT_DoPD is implemented, this field resets to the value in CTIDEVCTL.RCE.
  - When FEAT_DoPD is not implemented, this field resets to the value in EDECR.RCE.

OSUC, bit [0]

OS Unlock Catch debug event pending. Possible values of this field are:

OSUC	Meaning
0b0	Reading this means that an OS Unlock Catch debug event is not pending. Writing this means no action.
0b1	Reading this means that an OS Unlock Catch debug event is pending. Writing this clears the pending OS Unlock Catch debug event.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Accessing EDESR

If a request to clear a pending Halting debug event is received at or about the time when halting becomes allowed, it is CONSTRAINED UNPREDICTABLE whether the event is taken.

If Core power is removed while a Halting debug event is pending, it is lost. However, it might become pending again when the Core is powered back on and Cold reset.

EDESR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0x020	EDESR

Accessible as follows:

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register return an ERROR.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.2.27 EDITCTRL, External Debug Integration mode Control register

The EDITCTRL characteristics are:

Purpose

Enables the external debug to switch from its default mode into integration mode, where test software can control directly the inputs and outputs of the PE, for integration testing or topology detection.

Configuration

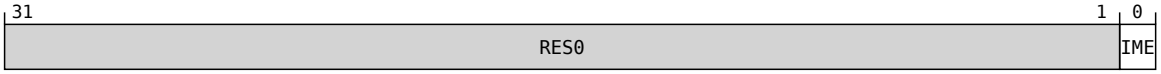
The power domain of EDITCTRL is IMPLEMENTATION DEFINED

Implementation of this register is OPTIONAL.

Attributes

EDITCTRL is a 32-bit register.

Field descriptions



Bits [31:1]

Reserved, RES0.

IME, bit [0]

Integration mode enable. When IME == 1, the device reverts to an integration mode to enable integration testing or topology detection.

IME	Meaning
0b0	Normal operation.
0b1	Integration mode enabled.

The integration mode behavior is IMPLEMENTATION DEFINED.

The following resets apply:

- Whichever power domain the register is implemented in, this field resets to 0.
- Otherwise, the value of this field is unchanged.

Accessing EDITCTRL

EDITCTRL can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xF00	EDITCTRL

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), accesses to this register are IMPLEMENTATION DEFINED.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.2.28 EDITR, External Debug Instruction Transfer Register

The EDITR characteristics are:

Purpose

Used in Debug state for passing instructions to the PE for execution.

Configuration

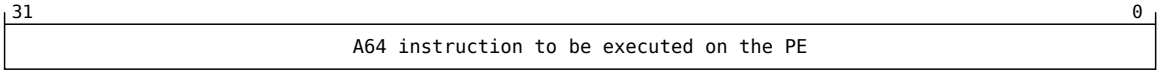
EDITR is in the Core power domain

Attributes

EDITR is a 32-bit register.

Field descriptions

When in AArch64 state:



A64_Instruction, bits [31:0]

A64 instruction to be executed on the PE.

Accessing EDITR

If EDSCR.ITE == 0 when the PE exits Debug state on receiving a Restart request trigger event, the behavior of any instruction issued through the ITR in Normal access mode that has not completed execution is CONSTRAINED UNPREDICTABLE, and must do one of the following:

- It must complete execution in Debug state before the PE executes the restart sequence.
- It must complete execution in Non-debug state before the PE executes the restart sequence.
- It must be abandoned. This means that the instruction does not execute. Any registers or memory accessed by the instruction are left in an UNKNOWN state.

EDITR ignores writes if the PE is in Non-debug state.

EDITR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0x084	EDITR

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), accesses to this register return an ERROR.
- When SoftwareLockStatus(), accesses to this register are WI.
- Otherwise, accesses to this register are WO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.2.29 EDLAR, External Debug Lock Access Register

The EDLAR characteristics are:

Purpose

Allows or disallows access to the external debug registers through a memory-mapped interface.

The optional Software Lock provides a lock to prevent memory-mapped writes to the debug registers. Use of this lock mechanism reduces the risk of accidental damage to the contents of the debug registers. It does not, and cannot, prevent all accidental or malicious damage.

Configuration

When FEAT_DoPD is implemented, EDLAR is in the Core power domain. Otherwise, EDLAR is in the Debug power domain

If FEAT_DoPD is implemented, Software Lock is not implemented by the architecturally-defined debug components of the PE.

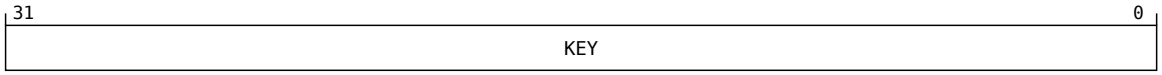
Software uses EDLAR to set or clear the lock, and EDLSR to check the current status of the lock.

Attributes

EDLAR is a 32-bit register.

Field descriptions

When Debug Software Lock is implemented:

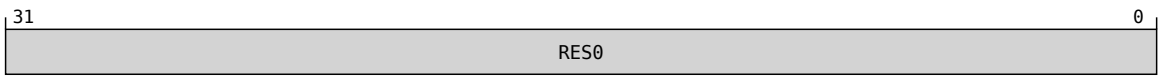


KEY, bits [31:0]

Lock Access control. Writing the key value `0xC5ACCE55` to this field unlocks the lock, enabling write accesses to this component's registers through a memory-mapped interface.

Writing any other value to this register locks the lock, disabling write accesses to this component's registers through a memory mapped interface.

Otherwise:



Otherwise

Bits [31:0]

Reserved, RES0.

Accessing EDLAR

EDLAR can be accessed through the memory-mapped interface:

Component	Offset	Instance
Debug	`0xFB0`	EDLAR

Accessible as follows:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are WO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.2.30 EDLSR, External Debug Lock Status Register

The EDLSR characteristics are:

Purpose

Indicates the current status of the software lock for external debug registers.

The optional Software Lock provides a lock to prevent memory-mapped writes to the debug registers. Use of this lock mechanism reduces the risk of accidental damage to the contents of the debug registers. It does not, and cannot, prevent all accidental or malicious damage.

Configuration

When FEAT_DoPD is implemented, EDLSR is in the Core power domain. Otherwise, EDLSR is in the Debug power domain

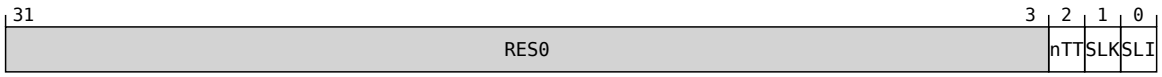
If FEAT_DoPD is implemented, Software Lock is not implemented by the architecturally-defined debug components of the PE.

Software uses EDLAR to set or clear the lock, and EDLSR to check the current status of the lock.

Attributes

EDLSR is a 32-bit register.

Field descriptions



Bits [31:3]

Reserved, RES0.

nTT, bit [2]

Not thirty-two bit access required. RAZ.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

SLK, bit [1]

When Debug Software Lock is implemented:

Software Lock status for this component. For an access to LSR that is not a memory-mapped access, or when Software Lock is not implemented, this field is RES0.

For memory-mapped accesses when Software Lock is implemented, possible values of this field are:

SLK	Meaning
0b0	Lock clear. Writes are permitted to this component's registers.
0b1	Lock set. Writes to this component's registers are ignored, and reads have no side effects.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_DoPD is implemented, this field resets to '1'.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a External debug reset:
  - When FEAT_DoPD is not implemented, this field resets to `'1'`.

Otherwise:

Reserved, RAZ.

SLI, bit [0]

Software Lock implemented. For an access to LSR that is not a memory-mapped access, this field is RAZ.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SLI	Meaning
`0b0`	Software Lock not implemented or not memory-mapped access.
`0b1`	Software Lock implemented and memory-mapped access.

Access to this field is RO.

Accessing EDLSR

EDLSR can be accessed through the memory-mapped interface:

Component	Offset	Instance
Debug	`0xFB4`	EDLSR

Accessible as follows:

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.2.31 EDPFR, External Debug Processor Feature Register

The EDPFR characteristics are:

Purpose

Provides information about implemented PE features.

For general information about the interpretation of the ID registers, see 'Principles of the ID scheme for fields in ID registers'.

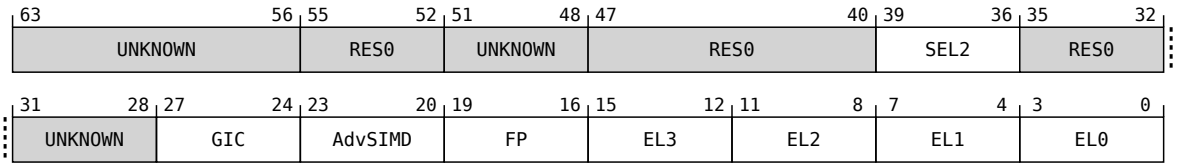
Configuration

There are no configuration notes.

Attributes

EDPFR is a 64-bit register.

Field descriptions



Bits [63:56]

Reserved, UNKNOWN.

Bits [55:52]

Reserved, RES0.

Bits [51:48]

Reserved, UNKNOWN.

Bits [47:40]

Reserved, RES0.

SEL2, bits [39:36]

Secure EL2.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SEL2	Meaning
0b0000	Secure EL2 is not implemented.
0b0001	Secure EL2 is implemented.

FEAT_SEL2 implements the functionality identified by the value 0b0001

All other values are reserved.

Access to this field is RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [35:32]

Reserved, RES0.

Bits [31:28]

Reserved, UNKNOWN.

GIC, bits [27:24]

System register GIC interface support.

The value of this field is an IMPLEMENTATION DEFINED choice of:

GIC	Meaning
0b0000	GIC CPU interface system registers not implemented.
0b0001	System register interface to versions 3.0 and 4.0 of the GIC CPU interface is supported.
0b0011	System register interface to version 4.1 of the GIC CPU interface is supported.

All other values are reserved.

In Armv8-R AArch64, this field returns the value of ID_AA64PFR0_EL1.GIC.

Access to this field is RO.

AdvSIMD, bits [23:20]

Advanced SIMD.

The value of this field is an IMPLEMENTATION DEFINED choice of:

AdvSIMD	Meaning
0b0000	Advanced SIMD is implemented, including support for the following SISD and SIMD operations: • Integer byte, halfword, word and doubleword element operations. • Single-precision and double-precision floating-point arithmetic. • Conversions between single-precision and half-precision data types, and double-precision and half-precision data types.
0b0001	As for 0b0000, and also includes support for half-precision floating-point arithmetic.
0b1111	Advanced SIMD is not implemented.

All other values are reserved.

This field must have the same value as the FP field.

The permitted values are:

- 0b0000 in an implementation with Advanced SIMD support, that does not include the FEAT_FP16 extension.
- 0b0001 in an implementation with Advanced SIMD support, that includes the FEAT_FP16 extension.
- 0b1111 in an implementation without Advanced SIMD support.

In Armv8-R AArch64, this field returns the value of ID_AA64PFR0_EL1.AdvSIMD.

Access to this field is RO.

FP, bits [19:16]

Floating-point.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FP	Meaning
0b0000	Floating-point is implemented, and includes support for: • Single-precision and double-precision floating-point types. • Conversions between single-precision and half-precision data types, and double-precision and half-precision data types.
0b0001	As for 0b0000, and also includes support for half-precision floating-point arithmetic.
0b1111	Floating-point is not implemented.

All other values are reserved.

This field must have the same value as the AdvSIMD field.

The permitted values are:

- 0b0000 in an implementation with floating-point support, that does not include the FEAT_FP16 extension.
- 0b0001 in an implementation with floating-point support, that includes the FEAT_FP16 extension.
- 0b1111 in an implementation without floating-point support.

In Armv8-R AArch64, this field returns the value of ID_AA64PFR0_EL1.FP.

Access to this field is RO.

EL3, bits [15:12]

AArch64 EL3 Exception level handling.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EL3	Meaning
0b0000	EL3 is not implemented or cannot be executed in AArch64 state.
0b0001	EL3 can be executed in AArch64 state only.
0b0010	EL3 can be executed in both Execution states.

When the value of EDAA32PFR.EL3 is nonzero, this field must be 0b0000.

All other values are reserved.

In an Armv8-R AArch64 implementation this field returns the value of 0b0000.

Access to this field is RO.

EL2, bits [11:8]

AArch64 EL2 Exception level handling.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EL2	Meaning
0b0000	EL2 is not implemented or cannot be executed in AArch64 state.
0b0001	EL2 can be executed in AArch64 state only.
0b0010	EL2 can be executed in both Execution states.

All other values are reserved.

In an Armv8-R AArch64 implementation this field returns the value of 0b0001.

Access to this field is RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EL1, bits [7:4]

AArch64 EL1 Exception level handling.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EL1	Meaning
0b0000	EL1 cannot be executed in AArch64 state.
0b0001	EL1 can be executed in AArch64 state only.
0b0010	EL1 can be executed in both Execution states.

All other values are reserved.

In Armv8-R AArch64, the only permitted values is 0b0001.

Access to this field is RO.

EL0, bits [3:0]

AArch64 EL0 Exception level handling.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EL0	Meaning
0b0000	EL0 cannot be executed in AArch64 state.
0b0001	EL0 can be executed in AArch64 state only.
0b0010	EL0 can be executed in both Execution states.

All other values are reserved.

In Armv8-R AArch64 implementation, the only permitted value is 0b0001.

Access to this field is RO.

Accessing EDPFR

EDPFR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xD20	EDPFR

Accessible as follows:

- When IsCorePowered() and !DoubleLockStatus(), accesses to this register are RO.
- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are IMPLEMENTATION DEFINED.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.2.32 EDPIDR0, External Debug Peripheral Identification Register 0

The EDPIDR0 characteristics are:

Purpose

Provides information to identify an external debug component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

When FEAT_DoPD is implemented, EDPIDR0 is in the Core power domain. Otherwise, EDPIDR0 is in the Debug power domain

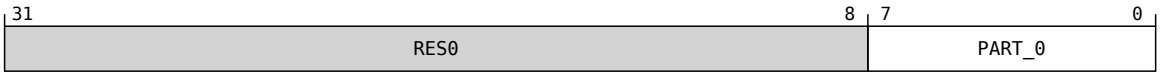
This register is required for CoreSight compliance.

Implementation of this register is OPTIONAL.

Attributes

EDPIDR0 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

PART_0, bits [7:0]

Part number, least significant byte.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing EDPIDR0

EDPIDR0 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xFE0	EDPIDR0

Accessible as follows:

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.2.33 EDPIDR1, External Debug Peripheral Identification Register 1

The EDPIDR1 characteristics are:

Purpose

Provides information to identify an external debug component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

When FEAT_DoPD is implemented, EDPIDR1 is in the Core power domain. Otherwise, EDPIDR1 is in the Debug power domain

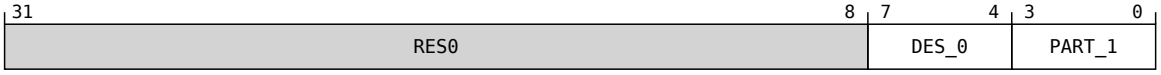
This register is required for CoreSight compliance.

Implementation of this register is OPTIONAL.

Attributes

EDPIDR1 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

DES_0, bits [7:4]

Designer, least significant nibble of JEP106 ID code. For Arm Limited, this field is `0b1011`.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

PART_1, bits [3:0]

Part number, most significant nibble.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing EDPIDR1

EDPIDR1 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0xFE4`	EDPIDR1

Accessible as follows:

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.2.34 EDPIDR2, External Debug Peripheral Identification Register 2

The EDPIDR2 characteristics are:

Purpose

Provides information to identify an external debug component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

When FEAT_DoPD is implemented, EDPIDR2 is in the Core power domain. Otherwise, EDPIDR2 is in the Debug power domain

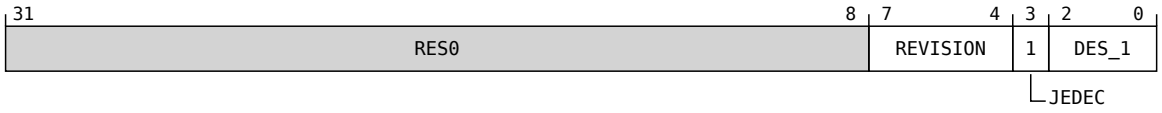
This register is required for CoreSight compliance.

Implementation of this register is OPTIONAL.

Attributes

EDPIDR2 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

REVISION, bits [7:4]

Part major revision. Parts can also use this field to extend Part number to 16-bits.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

JEDEC, bit [3]

Indicates a JEP106 identity code is used.

Reads as `0b1`

Access to this field is RO.

DES_1, bits [2:0]

Designer, most significant bits of JEP106 ID code. For Arm Limited, this field is `0b011`.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing EDPIDR2

EDPIDR2 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0xFE8`	EDPIDR2

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessible as follows:

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.2.35 EDPIDR3, External Debug Peripheral Identification Register 3

The EDPIDR3 characteristics are:

Purpose

Provides information to identify an external debug component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

When FEAT_DoPD is implemented, EDPIDR3 is in the Core power domain. Otherwise, EDPIDR3 is in the Debug power domain

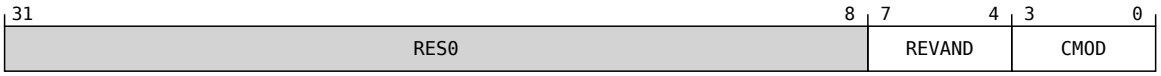
This register is required for CoreSight compliance.

Implementation of this register is OPTIONAL.

Attributes

EDPIDR3 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

REVAND, bits [7:4]

Part minor revision. Parts using EDPIDR2.REVISION as an extension to the Part number must use this field as a major revision number.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

CMOD, bits [3:0]

Customer modified. Indicates someone other than the Designer has modified the component.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing EDPIDR3

EDPIDR3 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xFEC	EDPIDR3

Accessible as follows:

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.2.36 EDPIDR4, External Debug Peripheral Identification Register 4

The EDPIDR4 characteristics are:

Purpose

Provides information to identify an external debug component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

When FEAT_DoPD is implemented, EDPIDR4 is in the Core power domain. Otherwise, EDPIDR4 is in the Debug power domain

This register is required for CoreSight compliance.

Implementation of this register is OPTIONAL.

Attributes

EDPIDR4 is a 32-bit register.

Field descriptions

Bits [31:8]

Reserved, RES0.

SIZE, bits [7:4]

Size of the component. Log_2 of the number of 4KB pages from the start of the component to the end of the component ID registers.

Reads as `0b0000`

Access to this field is RO.

DES_2, bits [3:0]

Designer, JEP106 continuation code, least significant nibble. For Arm Limited, this field is `0b0100`.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing EDPIDR4

EDPIDR4 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0xFD0`	EDPIDR4

Accessible as follows:

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.2.37 EDPRCR, External Debug Power/Reset Control Register

The EDPRCR characteristics are:

Purpose

Controls the PE functionality related to powerup, reset, and powerdown.

Configuration

When FEAT_DoPD is implemented, EDPRCR is in the Core power domain. Otherwise, EDPRCR contains fields that are in the Core power domain and fields that are in the Debug power domain

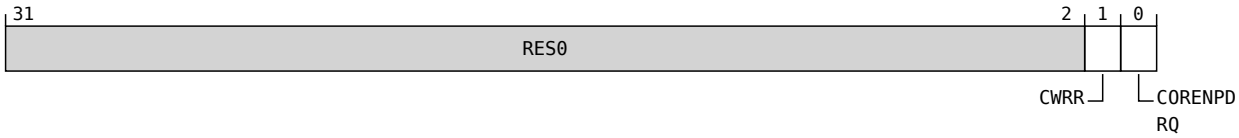
External register EDPRCR bit [0] is architecturally mapped to AArch64 System register DBGPRCR_EL1[0].

Attributes

EDPRCR is a 32-bit register.

Field descriptions

When FEAT_DoPD is implemented:



Bits [31:2]

Reserved, RES0.

CWRR, bit [1]

Warm reset request.

The extent of the reset is IMPLEMENTATION DEFINED, but must be one of:

- The request is ignored.
- Only this PE is Warm reset.
- This PE and other components of the system, possibly including other PEs, are Warm reset.

Arm deprecates use of this bit, and recommends that implementations ignore the request.

CWRR	Meaning
0b0	No action.
0b1	Request Warm reset.

This field is in the Core power domain

In an implementation that includes the recommended external debug interface, this bit drives the **DBGRSTREQ** signal.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Accessing this field has the following behavior:

- Access to this field is RAZ/WI if any of the following are true:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- OSLockStatus()
- SoftwareLockStatus()

- Access to this field is RAZ/WI if all of the following are true:
  - !ExternalSecureInvasiveDebugEnabled()
  - FEAT_Secure is implemented
- Access to this field is RAZ/WI if all of the following are true:
  - !ExternalInvasiveDebugEnabled()
  - FEAT_Secure is not implemented
- Otherwise, access to this field is WO/RAZ.

CORENPDRQ, bit [0]

Core no powerdown request. Requests emulation of powerdown.

This request is typically passed to an external power controller. This means that whether a request causes power up is dependent on the IMPLEMENTATION DEFINED nature of the system. The power controller must not allow the Core power domain to switch off while this bit is 1.

CORENPDRQ	Meaning
0b0	If the system responds to a powerdown request, it powers down Core power domain.
0b1	If the system responds to a powerdown request, it does not powerdown the Core power domain, but instead emulates a powerdown of that domain.

When this bit reads as UNKNOWN, the PE ignores writes to this bit.

This field is in the Core power domain, and permitted accesses to this field map to the DBGPRCR.CORENPDRQ and DBGPRCR_EL1.CORENPDRQ fields.

In an implementation that includes the recommended external debug interface, this bit drives the **DBGNOPWRDWN** signal.

It is IMPLEMENTATION DEFINED whether this bit is reset to the Cold reset value on exit from an IMPLEMENTATION DEFINED software-visible retention state. For more information about retention states, see 'Core power domain power states'.

Note

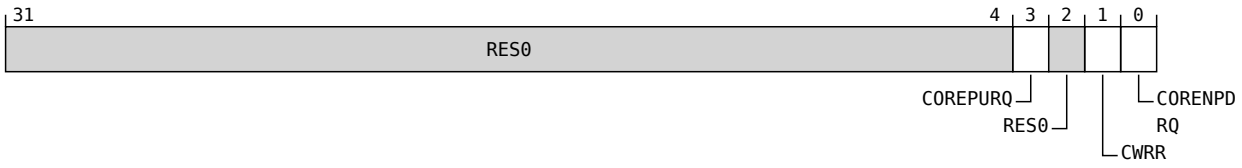
Writes to this bit are not prohibited by the IMPLEMENTATION DEFINED authentication interface. This means that a debugger can request emulation of powerdown regardless of whether invasive debug is permitted.

On a Cold reset, if the powerup request is implemented and the powerup request has been asserted, this field is an IMPLEMENTATION DEFINED choice of 0 or 1. If the powerup request is not asserted, this field is set to 0.

Accessing this field has the following behavior:

- When OSLockStatus(), access to this field is UNKNOWN/WI.
- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is RW.

Otherwise:



Bits [31:4]

Reserved, RES0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

COREPURQ, bit [3]

Core powerup request. Allows a debugger to request that the power controller power up the core, enabling access to the debug register in the Core power domain, and that the power controller emulates powerdown.

This request is typically passed to an external power controller. This means that whether a request causes power up is dependent on the IMPLEMENTATION DEFINED nature of the system. The power controller must not allow the Core power domain to switch off while this bit is 1.

COREPURQ	Meaning
0b0	Do not request power up of the Core power domain.
0b1	Request power up of the Core power domain, and emulation of powerdown.

In an implementation that includes the recommended external debug interface, this bit drives the **DBGPWRUPREQ** signal.

This field is in the Debug power domain and can be read and written when the Core power domain is powered off.

Note

Writes to this bit are not prohibited by the IMPLEMENTATION DEFINED authentication interface. This means that a debugger can request emulation of powerdown regardless of whether invasive debug is permitted.

The reset behavior of this field is:

- On a External debug reset, this field resets to '0'.

Accessing this field has the following behavior:

- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is RW.

Bit [2]

Reserved, RES0.

CWRR, bit [1]

Warm reset request.

The extent of the reset is IMPLEMENTATION DEFINED, but must be one of:

- The request is ignored.
- Only this PE is Warm reset.
- This PE and other components of the system, possibly including other PEs, are Warm reset.

Arm deprecates use of this bit, and recommends that implementations ignore the request.

CWRR	Meaning
0b0	No action.
0b1	Request Warm reset.

This field is in the Core power domain

In an implementation that includes the recommended external debug interface, this bit drives the **DBGRSTREQ** signal.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing this field has the following behavior:

- Access to this field is RAZ/WI if any of the following are true:
  - !IsCorePowered()
  - DoubleLockStatus()
  - OSLockStatus()
  - SoftwareLockStatus()
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_Secure is implemented
  - !ExternalSecureInvasiveDebugEnabled()
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_Secure is not implemented
  - !ExternalInvasiveDebugEnabled()
- Otherwise, access to this field is WO/RAZ.

CORENPDRQ, bit [0]

Core no powerdown request. Requests emulation of powerdown.

This request is typically passed to an external power controller. This means that whether a request causes power up is dependent on the IMPLEMENTATION DEFINED nature of the system. The power controller must not allow the Core power domain to switch off while this bit is 1.

CORENPDRQ	Meaning
0b0	If the system responds to a powerdown request, it powers down Core power domain.
0b1	If the system responds to a powerdown request, it does not powerdown the Core power domain, but instead emulates a powerdown of that domain.

When this bit reads as UNKNOWN, the PE ignores writes to this bit.

This field is in the Core power domain, and permitted accesses to this field map to the DBGPRCR.CORENPDRQ and DBGPRCR_EL1.CORENPDRQ fields.

In an implementation that includes the recommended external debug interface, this bit drives the **DBGNOPWRDWN** signal.

It is IMPLEMENTATION DEFINED whether this bit is reset to the value of EDPRCR.COREPURQ on exit from an IMPLEMENTATION DEFINED software-visible retention state. For more information about retention states, see 'Core power domain power states'.

Note

Writes to this bit are not prohibited by the IMPLEMENTATION DEFINED authentication interface. This means that a debugger can request emulation of powerdown regardless of whether invasive debug is permitted.

The reset behavior of this field is:

- On a Cold reset, this field resets to the value in EDPRCR.COREPURQ.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if any of the following are true:
  - !IsCorePowered()
  - DoubleLockStatus()
  - OSLockStatus()
- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing EDPRCR

On permitted accesses to the register, other access controls affect the behavior of some fields. See the field descriptions for more information.

EDPRCR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0x310	EDPRCR

Accessible as follows:

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.2.38 EDPRSR, External Debug Processor Status Register

The EDPRSR characteristics are:

Purpose

Holds information about the reset and powerdown state of the PE.

Configuration

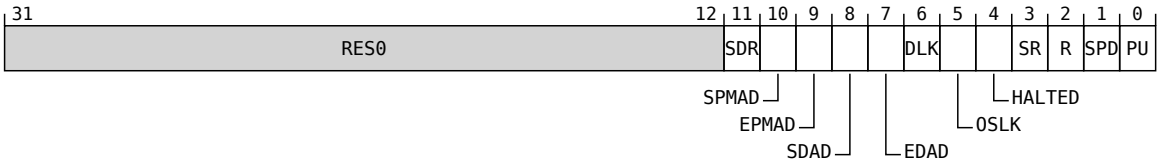
When FEAT_DoPD is implemented, EDPRSR is in the Core power domain. Otherwise, EDPRSR contains fields that are in the Core power domain and fields that are in the Debug power domain

If FEAT_DoPD is implemented, then all fields in this register are in the Core power domain.

Attributes

EDPRSR is a 32-bit register.

Field descriptions



Bits [31:12]

Reserved, RES0.

SDR, bit [11]

Sticky Debug Restart. Set to 1 when the PE exits Debug state.

SDR	Meaning
0b0	The PE has not restarted since EDPRSR was last read.
0b1	The PE has restarted since EDPRSR was last read.

Note

If a reset occurs when the PE is in Debug state, the PE exits Debug state. SDR is UNKNOWN on Warm reset, meaning a debugger must also use the SR bit to determine whether the PE has left Debug state.

If the Core power domain is powered up, then following a read of EDPRSR:

- If FEAT_DoubleLock is not implemented or `DoubleLockStatus()` == FALSE, this bit clears to 0.
- If FEAT_DoubleLock is implemented and `DoubleLockStatus()` == TRUE, it is CONSTRAINED UNPREDICTABLE whether this bit clears to 0 or is unchanged.

This field is in the Core power domain.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if all of the following are true:
  - FEAT_DoPD is not implemented

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- !IsCorePowered()
- When DoubleLockStatus(), access to this field is UNKNOWN/WI.
- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is RC/WI.

SPMAD, bit [10]

When FEAT_Debugv8p4 is implemented and FEAT_PMUv3_EXT is implemented:

Sticky EPMAD error. Set to 1 if an external debug interface access to a Performance Monitors register returns an error because `AllowExternalPMUAccess()` == FALSE.

SPMAD	Meaning
`0b0`	No Non-secure external debug interface accesses to the external Performance Monitors registers have failed because `AllowExternalPMUAccess()` == FALSE for the access since EDPRSR was last read.
`0b1`	At least one Non-secure external debug interface access to the external Performance Monitors register has failed and returned an error because `AllowExternalPMUAccess()` == FALSE for the access since EDPRSR was last read.

If the Core power domain is powered up, then following a read of EDPRSR:

- If FEAT_DoubleLock is not implemented or `DoubleLockStatus()` == FALSE, this bit clears to 0.
- If FEAT_DoubleLock is implemented and `DoubleLockStatus()` == TRUE, it is CONSTRAINED UNPREDICTABLE whether this bit clears to 0 or is unchanged.

This field is in the Core power domain.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if all of the following are true:
  - FEAT_DoPD is not implemented
  - !IsCorePowered()
- When DoubleLockStatus(), access to this field is UNKNOWN/WI.
- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is RC/WI.

When FEAT_PMUv3_EXT is implemented:

Sticky EPMAD error.

SPMAD	Meaning
`0b0`	No external debug interface accesses to the Performance Monitors registers have failed because `AllowExternalPMUAccess()` == FALSE since EDPRSR was last read.
`0b1`	At least one external debug interface access to the Performance Monitors registers has failed and returned an error because `AllowExternalPMUAccess()` == FALSE since EDPRSR was last read.

If the Core power domain is powered up, then, following a read of EDPRSR:

- If FEAT_DoubleLock is not implemented or `DoubleLockStatus()` == FALSE, this bit clears to 0.
- If FEAT_DoubleLock is implemented, and `DoubleLockStatus()` == TRUE, it is CONSTRAINED UNPREDICTABLE whether this bit clears to 0 or is unchanged.

This field is in the Core power domain.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if all of the following are true:
  - FEAT_DoPD is not implemented
  - !IsCorePowered()
- When DoubleLockStatus(), access to this field is UNKNOWN/WI.
- When OSLockStatus(), access to this field is UNKNOWN/WI.
- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is RC/WI.

Otherwise:

Reserved, RES0.

EPMAD, bit [9]

When FEAT_Debugv8p4 is implemented and FEAT_PMUv3_EXT is implemented:

External Performance Monitors Access Disable status. Reports whether Non-secure access to Performance Monitor registers by an external debugger is prohibited by the MDCR_EL3.EPMAD control.

See MDCR_EL3.EPMAD for the list of affected external Performance Monitor registers.

EPMAD	Meaning
0b0	External Non-secure access to Performance Monitor registers not affected. AllowExternalPMUAccess() == TRUE for a Non-secure access.
0b1	External Non-secure access to affected Performance Monitor registers is prohibited. AllowExternalPMUAccess() == FALSE for a Non-secure access.

This field is in the Core power domain.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if any of the following are true:
  - All of the following are true:
    - FEAT_DoPD is not implemented
    - !IsCorePowered()
  - EDPRSR.R == '1'
- When DoubleLockStatus(), access to this field is UNKNOWN/WI.
- Otherwise, access to this field is RO.

When FEAT_PMUv3_EXT is implemented:

External Performance Monitors Access Disable status. Reports whether access to Performance Monitor registers by an external debugger is prohibited by the MDCR_EL3.EPMAD control.

See MDCR_EL3.EPMAD for the list of affected external Performance Monitor registers.

EPMAD	Meaning
0b0	External access to Performance Monitor registers not affected. AllowExternalPMUAccess() == TRUE.
0b1	External access to affected Performance Monitor registers is prohibited. AllowExternalPMUAccess() == FALSE.

This field is in the Core power domain.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if any of the following are true:
  - All of the following are true:
    - FEAT_DoPD is not implemented

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— !IsCorePowered()
- EDPRSR.R == '1'

- When DoubleLockStatus(), access to this field is UNKNOWN/WI.
- When OSLockStatus(), access to this field is UNKNOWN/WI.
- Otherwise, access to this field is RO.

When FEAT_PMUv3 is implemented:

Reserved, UNKNOWN.

Otherwise:

Reserved, RES0.

SDAD, bit [8]

When FEAT_Debugv8p4 is implemented:

Sticky EDAD error. Set to 1 if an external debug interface access to a debug register returns an error because `AllowExternalDebugAccess()` == FALSE.

SDAD	Meaning
`0b0`	No Non-secure external debug interface accesses to the debug registers have failed because `AllowExternalDebugAccess()` == FALSE for the access since EDPRSR was last read.
`0b1`	At least one Non-secure external debug interface access to the debug registers has failed and returned an error because `AllowExternalDebugAccess()` == FALSE for the access since EDPRSR was last read.

If the Core power domain is powered up, then, following a read of EDPRSR:

- If FEAT_DoubleLock is not implemented or `DoubleLockStatus()` == FALSE, this bit clears to 0.
- If FEAT_DoubleLock is implemented and `DoubleLockStatus()` == TRUE, it is CONSTRAINED UNPREDICTABLE whether this bit clears to 0 or is unchanged.

This field is in the Core power domain.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if any of the following are true:
  - All of the following are true:
    — FEAT_DoPD is not implemented
    — !IsCorePowered()
  - EDPRSR.R == '1'
- When DoubleLockStatus(), access to this field is UNKNOWN/WI.
- Otherwise, access to this field is RO.

Otherwise:

Sticky EDAD error. Set to 1 if an external debug interface access to a debug register returns an error because `AllowExternalDebugAccess()` == FALSE.

SDAD	Meaning
`0b0`	No external debug interface accesses to the debug registers have failed because `AllowExternalDebugAccess()` == FALSE since EDPRSR was last read.
`0b1`	At least one external debug interface access to the debug registers has failed and returned an error because `AllowExternalDebugAccess()` == FALSE since EDPRSR was last read.

If the Core power domain is powered up, then, following a read of EDPRSR:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If FEAT_DoubleLock is not implemented or `DoubleLockStatus()` == FALSE, this bit clears to 0.
- If FEAT_DoubleLock is implemented and `DoubleLockStatus()` == TRUE, it is CONSTRAINED UNPREDICTABLE whether this bit clears to 0 or is unchanged.

This field is in the Core power domain.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if any of the following are true:
  - All of the following are true:
    — FEAT_DoPD is not implemented
    — !IsCorePowered()
  - EDPRSR.R == '1'
- When DoubleLockStatus(), access to this field is UNKNOWN/WI.
- Access to this field is UNKNOWN/WI if all of the following are true:
  - OSLockStatus()
  - external debug writes to OSLAR_EL1 do not return an error when AllowExternalDebugAccess(addrdesc) == FALSE
- Otherwise, access to this field is RO.

EDAD, bit [7]

When FEAT_Debugv8p4 is implemented:

External Debug Access Disable status. Reports whether Non-secure access to breakpoint registers, watchpoint registers, and OSLAR_EL1 by an external debugger is prohibited by the MDCR_EL3.EDAD control.

See MDCR_EL3.EDAD for the list of affected external debug registers.

EDAD	Meaning
`0b0`	External Non-secure access to debug registers not affected. `AllowExternalDebugAccess()` == TRUE for a Non-secure access.
`0b1`	External Non-secure access to affected debug registers is prohibited. `AllowExternalDebugAccess()` == FALSE for a Non-secure access.

This field is in the Core power domain.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if any of the following are true:
  - All of the following are true:
    — FEAT_DoPD is not implemented
    — !IsCorePowered()
  - EDPRSR.R == '1'
- When DoubleLockStatus(), access to this field is UNKNOWN/WI.
- Otherwise, access to this field is RO.

When FEAT_Debugv8p2 is implemented:

External Debug Access Disable status. Reports whether access to breakpoint registers, watchpoint registers, and OSLAR_EL1 by an external debugger is prohibited by the MDCR_EL3.EDAD control.

See MDCR_EL3.EDAD for the list of affected external debug registers.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EDAD	Meaning
0b0	External access to debug registers not affected. AllowExternalDebugAccess() == TRUE.
0b1	External access to affected debug registers is prohibited. AllowExternalDebugAccess() == FALSE.

This field is in the Core power domain.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if any of the following are true:
  - All of the following are true:
    - FEAT_DoPD is not implemented
    - !IsCorePowered()
  - EDPRSR.R == '1'
- When DoubleLockStatus(), access to this field is UNKNOWN/WI.
- Access to this field is UNKNOWN/WI if all of the following are true:
  - OSLockStatus()
  - external debug writes to OSLAR_EL1 do not return an error when AllowExternalDebugAccess(addrdesc) == FALSE
- Otherwise, access to this field is RO.

Otherwise:

External Debug Access Disable status. Reports whether access to breakpoint registers, watchpoint registers, and optionally OSLAR_EL1 by an external debugger is prohibited by the MDCR_EL3.EDAD control.

See MDCR_EL3.EDAD for the list of affected external debug registers.

EDAD	Meaning
0b0	External access to debug registers not affected. AllowExternalDebugAccess() == TRUE.
0b1	External access to affected debug registers is prohibited. AllowExternalDebugAccess() == FALSE.

This field is in the Core power domain.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if any of the following are true:
  - All of the following are true:
    - FEAT_DoPD is not implemented
    - !IsCorePowered()
  - EDPRSR.R == '1'
- When DoubleLockStatus(), access to this field is UNKNOWN/WI.
- Otherwise, access to this field is RO.

DLK, bit [6]

When FEAT_Debugv8p4 is implemented:

This field is RES0.

When FEAT_Debugv8p2 is implemented and FEAT_DoubleLock is implemented:

Double Lock. From Armv8.2, use of this field is deprecated.

This field is in the Core power domain.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if any of the following are true:
  - !IsCorePowered()
  - DoubleLockStatus()

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Otherwise, access to this field is RAZ/WI.

When FEAT_DoubleLock is implemented:

Double Lock.

This field returns the result of the pseudocode function `DoubleLockStatus()`.

If the Core power domain is powered up and `DoubleLockStatus()` == TRUE, it is IMPLEMENTATION DEFINED whether:

- EDPRSR.PU reads as 1, EDPRSR.DLK reads as 1, and EDPRSR.SPD is UNKNOWN.
- EDPRSR.PU reads as 0, EDPRSR.DLK is UNKNOWN, and EDPRSR.SPD reads as 0.

This field is in the Core power domain.

DLK	Meaning
0b0	`DoubleLockStatus()` returns FALSE.
0b1	`DoubleLockStatus()` returns TRUE and the Core power domain is powered up.

Accessing this field has the following behavior:

- When !IsCorePowered(), access to this field is UNKNOWN/WI.
- Otherwise, access to this field is RO.

Otherwise:

Reserved, RES0.

OSLK, bit [5]

OS Lock status bit.

A read of this bit returns the value of OSLSR_EL1.OSLK.

This field is in the Core power domain.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if any of the following are true:
  - All of the following are true:
    - FEAT_DoPD is not implemented
    - !IsCorePowered()
  - EDPRSR.R == '1'
- When DoubleLockStatus(), access to this field is UNKNOWN/WI.
- Otherwise, access to this field is RO.

HALTED, bit [4]

Halted status bit.

HALTED	Meaning
0b0	PE is in Non-debug state.
0b1	PE is in Debug state.

This field is in the Core power domain.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if all of the following are true:
  - FEAT_DoPD is not implemented

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- !IsCorePowered()
- Otherwise, access to this field is RO.

SR, bit [3]

Sticky core Reset status bit.

SR	Meaning
0b0	The non-debug logic of the PE is not in reset state and has not been reset since the last time EDPRSR was read.
0b1	The non-debug logic of the PE is in reset state or has been reset since the last time EDPRSR was read.

If EDPRSR.PU reads as 1 and EDPRSR.R reads as 0, which means that the Core power domain is in a powerup state and that the non-debug logic of the PE is not in reset state, then following a read of EDPRSR:

- If FEAT_DoubleLock is not implemented or `DoubleLockStatus()` == FALSE, this bit clears to 0.
- If FEAT_DoubleLock is implemented and `DoubleLockStatus()` == TRUE, it is CONSTRAINED UNPREDICTABLE whether this bit clears to 0 or is unchanged.

This field is in the Core power domain.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'1'`.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if all of the following are true:
  - FEAT_DoPD is not implemented
  - !IsCorePowered()
- When DoubleLockStatus(), access to this field is UNKNOWN/WI.
- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is RC/WI.

R, bit [2]

PE Reset status bit.

R	Meaning
0b0	The non-debug logic of the PE is not in reset state.
0b1	The non-debug logic of the PE is in reset state.

If FEAT_DoubleLock is implemented, the PE is in reset state, and the PE entered reset state with the OS Double Lock locked this bit has a CONSTRAINED UNPREDICTABLE value. For more information, see ‘EDPRSR.{DLK, R} and reset state’.

This field is in the Core power domain.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if any of the following are true:
  - All of the following are true:
    - FEAT_DoPD is not implemented
    - !IsCorePowered()
  - DoubleLockStatus()
- Otherwise, access to this field is RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SPD, bit [1]

Sticky core Powerdown status bit.

If FEAT_DoubleLock is implemented and `DoubleLockStatus()` == TRUE, then:

- If FEAT_Debugv8p2 is implemented, this bit reads as 0.
- If FEAT_Debugv8p2 is not implemented, this bit might read as 0 or 1.

For more information, see 'EDPRSR.{DLK, SPD, PU} and the Core power domain'.

SPD	Meaning
`0b0`	If EDPRSR.PU is 0, it is not known whether the state of the debug registers in the Core power domain is lost. If EDPRSR.PU is 1, the state of the debug registers in the Core power domain has not been lost.
`0b1`	The state of the debug registers in the Core power domain has been lost.

If the Core power domain is powered up, then, following a read of EDPRSR:

- If FEAT_DoubleLock is not implemented or `DoubleLockStatus()` == FALSE, this bit clears to 0.
- If FEAT_DoubleLock is implemented and `DoubleLockStatus()` == TRUE, it is CONSTRAINED UNPREDICTABLE whether this bit clears to 0 or is unchanged.

EDPRSR.{DLK, SPD, PU} describe whether registers in the Core power domain can be accessed, and whether their state has been lost since the last time the register was read. For more information, see 'EDPRSR.{DLK, SPD, PU} and the Core power domain'.

When FEAT_DoPD is not implemented and the Core power domain is in either retention or powerdown state, the value of EDPRSR.SPD is IMPLEMENTATION DEFINED. For more information, see 'EDPRSR.SPD when the Core domain is in either retention or powerdown state'.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'1'`.

Accessing this field has the following behavior:

- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_DoPD is not implemented
  - !IsCorePowered()
- Access to this field is UNKNOWN/WI if all of the following are true:
  - IsCorePowered()
  - DoubleLockStatus()
- Otherwise, access to this field is RC/WI.

PU, bit [0]

When FEAT_DoPD is implemented:

Core powerup status bit.

Access to this field is RAO/WI.

When FEAT_Debugv8p2 is implemented:

Core Powerup status bit. Indicates whether the debug registers in the Core power domain can be accessed.

PU	Meaning
`0b0`	Either the Core power domain is in a low-power or powerdown state, or FEAT_DoubleLock is implemented and `DoubleLockStatus()` == TRUE, meaning the debug registers in the Core power domain cannot be accessed.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PU	Meaning
0b1	The Core power domain is in a powerup state, and either FEAT_DoubleLock is not implemented or DoubleLockStatus() == FALSE, meaning the debug registers in the Core power domain can be accessed.

If FEAT_DoubleLock is implemented, the PE is in reset state, and the PE entered reset state with the OS Double Lock locked this bit has a CONSTRAINED UNPREDICTABLE value. For more information, see 'EDPRSR.{DLK, R} and reset state'

EDPRSR.{DLK, SPD, PU} describe whether registers in the Core power domain can be accessed, and whether their state has been lost since the last time the register was read. For more information, see 'EDPRSR.{DLK, SPD, PU} and the Core power domain'

Access to this field is RO.

Otherwise:

Core Powerup status bit. Indicates whether the debug registers in the Core power domain can be accessed.

PU	Meaning
0b0	Core power domain is in a low-power or powerdown state where the debug registers in the Core power domain cannot be accessed.
0b1	Core power domain is in a powerup state where the debug registers in the Core power domain can be accessed.

If FEAT_DoubleLock is implemented, the PE is in reset state, and the PE entered reset state with the OS Double Lock locked this bit has a CONSTRAINED UNPREDICTABLE value. For more information see 'EDPRSR.{DLK, R} and reset state'

EDPRSR.{DLK, SPD, PU} describe whether registers in the Core power domain can be accessed, and whether their state has been lost since the last time the register was read. For more information, see 'EDPRSR.{DLK, SPD, PU} and the Core power domain'.

When the Core power domain is powered-up and DoubleLockStatus() == TRUE, then the value of EDPRSR.PU is IMPLEMENTATION DEFINED. See the description of the DLK bit for more information.

If FEAT_DoubleLock is implemented, the Core power domain is powered up, and DoubleLockStatus() == TRUE, it is IMPLEMENTATION DEFINED whether this bit reads as 0 or 1.

Access to this field is RO.

Accessing EDPRSR

On permitted accesses to the register, other access controls affect the behavior of some fields. See the field descriptions for more information.

If the Core power domain is powered up (EDPRSR.PU == 1), then following a read of EDPRSR:

- If FEAT_DoubleLock is not implemented or DoubleLockStatus() == FALSE, then:
  - EDPRSR.{SDR, SPMAD, SDAD, SPD} are cleared to 0.
  - EDPRSR.SR is cleared to 0 if the non-debug logic of the PE is not in reset state (EDPRSR.R == 0).
- If FEAT_DoubleLock is implemented and DoubleLockStatus() == TRUE, it is CONSTRAINED UNPREDICTABLE whether or not this clearing occurs.

If FEAT_DoPD is not implemented and the Core power domain is powered down (EDPRSR.PU == 0), then:

- EDPRSR.{SDR, SPMAD, SDAD, SR} are all UNKNOWN, and are either reset or restored on being powered up.
- EDPRSR.SPD is not cleared following a read of EDPRSR. See the SPD bit description for more information.

The clearing of bits is an indirect write to EDPRSR.

EDPRSR can be accessed through the external debug interface:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Offset	Instance
Debug	0x314	EDPRSR

Accessible as follows:

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.2.39 EDRCR, External Debug Reserve Control Register

The EDRCR characteristics are:

Purpose

This register is used to allow imprecise entry to Debug state and clear sticky bits in EDSCR.

Configuration

EDRCR is in the Core power domain

Attributes

EDRCR is a 32-bit register.

Field descriptions

Bits [31:5]

Reserved, RES0.

CBRRQ, bit [4]

Allow imprecise entry to Debug state. The actions on writing to this bit are:

CBRRQ	Meaning
0b0	No action.
0b1	Allow imprecise entry to Debug state, for example by canceling pending bus accesses.

Setting this bit to 1 allows a debugger to request imprecise entry to Debug state. An External Debug Request debug event must be pending before the debugger sets this bit to 1.

This feature is optional. If this feature is not implemented, writes to this bit are ignored.

CSPA, bit [3]

Clear Sticky Pipeline Advance. This bit is used to clear the EDSCR.PipeAdv bit to 0. The actions on writing to this bit are:

CSPA	Meaning
0b0	No action.
0b1	Clear the EDSCR.PipeAdv bit to 0.

CSE, bit [2]

Clear Sticky Error. Used to clear the EDSCR cumulative error bits to 0. The actions on writing to this bit are:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CSE	Meaning
0b0	No action.
0b1	Clear the EDSCR.{TXU, RXO, ERR} bits, and, if the PE is in Debug state, the EDSCR.ITO bit, to 0.

Bits [1:0]

Reserved, RES0.

Accessing EDRCR

EDRCR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0x090	EDRCR

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), accesses to this register return an ERROR.
- When SoftwareLockStatus(), accesses to this register are WI.
- Otherwise, accesses to this register are WO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.2.40 EDSCR, External Debug Status and Control Register

The EDSCR characteristics are:

Purpose

Main control register for the debug implementation.

Configuration

EDSCR is in the Core power domain

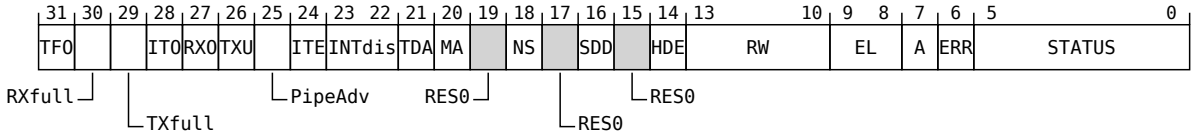
External register EDSCR bits [31:29, 27:26, 23:21, 19, 14, 6] are architecturally mapped to AArch64 System register MDSCR_EL1[31:29, 27:26, 23:21, 19, 14, 6].

External register EDSCR bits [30:29] are architecturally mapped to AArch64 System register MDCCSR_EL0[30:29].

Attributes

EDSCR is a 32-bit register.

Field descriptions



TFO, bit [31]

When FEAT_TRF is implemented:

Trace Filter Override. Overrides the Trace Filter controls allowing the external debugger to trace any visible Exception level.

TFO	Meaning
0b0	Trace Filter controls are not affected.
0b1	Trace Filter controls in TRFCR_EL1 and TRFCR_EL2 are ignored.

When OSLSR_EL1.OSLK is 1, this bit can be indirectly read and written through MDSCR_EL1.

This bit is ignored by the PE when ExternalSecureNoninvasiveDebugEnabled() is FALSE and the Effective value of MDCR_EL3.STE is 1.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Otherwise:

Reserved, RES0.

RXfull, bit [30]

DTRRX full.

When OSLSR_EL1.OSLK is 1, this bit can be indirectly read and written through MDSCR_EL1.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Access to this field is RO.

TXfull, bit [29]

DTRTX full.

When OSLSR_EL1.OSLK is 1, this bit can be indirectly read and written through MDSCR_EL1.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Access to this field is RO.

ITO, bit [28]

ITR overrun. Set to 0 on entry to Debug state.

Accessing this field has the following behavior:

- When the PE is in Non-debug state, access to this field is UNKNOWN/WI.
- Otherwise, access to this field is RO.

RXO, bit [27]

DTRRX overrun.

When OSLSR_EL1.OSLK is 1, this bit can be indirectly read and written through MDSCR_EL1.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Access to this field is RO.

TXU, bit [26]

DTRTX underrun.

When OSLSR_EL1.OSLK is 1, this bit can be indirectly read and written through MDSCR_EL1.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Access to this field is RO.

PipeAdv, bit [25]

Pipeline Advance. Indicates that software execution is progressing.

PipeAdv	Meaning
0b0	No progress has been made by the PE since the last time this field was cleared to zero by writing 1 to EDRCR.CSPA.
0b1	Progress has been made by the PE since the last time this field was cleared to zero by writing 1 to EDRCR.CSPA.

The architecture does not define precisely when this field is set to 1. It requires only that this happen periodically in Non-debug state to indicate that software execution is progressing. For example, a PE might set this field to 1 each time the PE retires one or more instructions, or at periodic intervals during the progression of an instruction.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ITE, bit [24]

ITR empty.

Accessing this field has the following behavior:

- When the PE is in Non-debug state, access to this field is UNKNOWN/WI.
- Otherwise, access to this field is RO.

INTdis, bits [23:22]

When FEAT_Debugv8p4 is implemented:

Interrupt and SError exception disable. Disables taking interrupts and SError exceptions in Non-debug state.

INTdis	Meaning
0b00	Masking of interrupts and SError exceptions is controlled by PSTATE and interrupt routing controls.
0b01	If ExternalInvasiveDebugEnabled() is TRUE, then all interrupts and SError exceptions taken to Non-secure state are masked. If ExternalSecureInvasiveDebugEnabled() is TRUE, then all interrupts and SError exceptions taken to Secure state are masked.

Note

This control affects both physical and virtual interrupts and SError exceptions.

When OSLSR_EL1.OSLK is 1, this field can be indirectly read and written through the following System registers:

- MDSCR_EL1.

The Effective value of this field is 0b00 when ExternalInvasiveDebugEnabled() is FALSE.

When FEAT_Debugv8p4 is implemented, bit[23] of this register is RES0.

The reset behavior of this field is:

- On a Cold reset, this field resets to '00'.

Otherwise:

Interrupt and SError exception disable. Disables taking interrupts and SError exceptions in Non-debug state.

INTdis	Meaning
0b00	Masking of interrupts and SError exceptions is controlled by PSTATE and interrupt routing controls.
0b01	If ExternalInvasiveDebugEnabled() is TRUE, then all interrupts and SError exceptions taken to Non-secure EL1 are masked.
0b10	If ExternalInvasiveDebugEnabled() is TRUE, then all interrupts and SError exceptions taken to Non-secure state are masked. If ExternalSecureInvasiveDebugEnabled() is TRUE, then all interrupts and SError exceptions taken to Secure EL1 are masked.
0b11	If ExternalInvasiveDebugEnabled() is TRUE, then all interrupts and SError exceptions taken to Non-secure state are masked. If ExternalSecureInvasiveDebugEnabled() is TRUE, then all interrupts and SError exceptions taken to Secure state are masked.

Note

This control affects both physical and virtual interrupts and SError exceptions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When OSLSR_EL1.OSLK is 1, this field can be indirectly read and written through the following System registers:

- MDSCR_EL1.

The Effective value of this field is 0b00 when ExternalInvasiveDebugEnabled() is FALSE.

Support for the values 0b01 and 0b10 is IMPLEMENTATION DEFINED. If these values are not supported, they are reserved. If programmed with a reserved value, the PE behaves as if EDSCR.INTdis has been programmed with a defined value, other than for a direct read of EDSCR, and the value returned by a read of EDSCR.INTdis is UNKNOWN.

The reset behavior of this field is:

- On a Cold reset, this field resets to '00'.

TDA, bit [21]

Traps accesses to the following debug System registers:

- AArch64: DBGBCR<n>_EL1, DBGBVR<n>_EL1, DBGWCR<n>_EL1, DBGWVR<n>_EL1.

TDA	Meaning
0b0	Accesses to debug System registers do not generate a Software Access Debug event.
0b1	Accesses to debug System registers generate a Software Access Debug event, if OSLSR_EL1.OSLK is 0 and if halting is allowed.

When OSLSR_EL1.OSLK is 1, this bit can be indirectly read and written through MDSCR_EL1.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

MA, bit [20]

Memory access mode. Controls the use of Memory access mode for accessing ITR and the DCC. This bit is ignored if in Non-debug state and set to zero on entry to Debug state.

Possible values of this field are:

MA	Meaning
0b0	Normal access mode.
0b1	Memory access mode.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Bit [19]

Reserved, RES0.

NS, bit [18]

Non-secure status. In Debug state, gives the current Security state:

NS	Meaning
0b0	Secure state.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NS	Meaning
0b1	Non-secure state.

Accessing this field has the following behavior:

- When the PE is in Non-debug state, access to this field is UNKNOWN/WI.
- Otherwise, access to this field is RO.

Bit [17]

Reserved, RES0.

SDD, bit [16]

When FEAT_Secure is implemented:

Secure debug disabled.

On entry to Debug state, SDD is set to 0.

In Debug state, the value of the SDD bit does not change, even if `ExternalSecureInvasiveDebugEnabled()` changes.

In Non-debug state, SDD returns the inverse of `ExternalSecureInvasiveDebugEnabled()`. If the authentication signals that control `ExternalSecureInvasiveDebugEnabled()` change, then a context synchronization event is required to guarantee their effect.

Access to this field is RO.

Otherwise:

Reserved, RES1.

Bit [15]

Reserved, RES0.

HDE, bit [14]

Halting debug enable.

HDE	Meaning
0b0	Halting disabled for Breakpoint, Watchpoint and Halt Instruction debug events.
0b1	Halting enabled for Breakpoint, Watchpoint and Halt Instruction debug events.

When OSLSR_EL1.OSLK is 1, this bit can be indirectly read and written through MDSCR_EL1.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

RW, bits [13:10]

Exception level Execution state status. In Debug state, each bit gives the current Execution state of each Exception level.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RW	Meaning
0b1111	Any of the following: • The PE is in Non-debug state. • The PE is at EL0 using AArch64. • The PE is not at EL0, and EL1 is using AArch64. If implemented and enabled in the current Security state, EL2 and EL3 are using AArch64.

Accessing this field has the following behavior:

- When the PE is in Non-debug state, access to this field is RAO/WI.
- Otherwise, access to this field is RO.

EL, bits [9:8]

Exception level. In Debug state, gives the current Exception level of the PE.

Accessing this field has the following behavior:

- When the PE is in Non-debug state, access to this field is RAZ/WI.
- Otherwise, access to this field is RO.

A, bit [7]

SError exception pending. In Debug state, indicates whether an SError exception is pending:

- If EL2 is enabled in the current Security state, the PE is executing at EL0 or EL1, and HCR_EL2.{AMO, TGE} is {1, 0}, a virtual SError exception.
- Otherwise, if FEAT_E3DSE is implemented, the PE is executing at EL0, EL1, or EL2, and SCR_EL3.EnDSE is 1, a delegated SError exception.
- Otherwise, a physical SError exception.

A	Meaning
0b0	No SError exception pending.
0b1	SError exception pending.

A debugger can read EDSCR to check whether an SError exception is pending without having to execute further instructions. A pending SError might indicate data from target memory is corrupted.

Accessing this field has the following behavior:

- When the PE is in Non-debug state, access to this field is UNKNOWN/WI.
- Otherwise, access to this field is RO.

ERR, bit [6]

Cumulative error flag. This bit is set to 1 following exceptions in Debug state and on any signaled overrun or underrun on the DTR or EDITR.

When OSLSR_EL1.OSLK is 1, this bit can be indirectly read and written through MDSCR_EL1.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Access to this field is RO.

STATUS, bits [5:0]

On entering Debug state, the PE sets this field to indicate the reason for halting.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

STATUS	Meaning
0b000001	PE is restarting, exiting Debug state.
0b000010	PE is in Non-debug state.
0b000111	Breakpoint.
0b010011	External debug request.
0b011011	Halting step, normal.
0b011111	Halting step, exclusive.
0b100011	OS Unlock Catch.
0b100111	Reset Catch.
0b101011	Watchpoint.
0b101111	HLT instruction.
0b110011	Software access to debug register.
0b110111	Exception Catch.
0b111011	Halting step, no syndrome.

All other values are reserved.

The PE resets into Non-debug state. However, if the PE enters Debug state immediately after reset, then the reset value is overwritten with the reason for halting.

The reset behavior of this field is:

- On a Cold reset, this field resets to '000010'.

Access to this field is RO.

Accessing EDSCR

EDSCR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0x088	EDSCR

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), accesses to this register return an ERROR.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.2.41 EDWAR, External Debug Watchpoint Address Register

The EDWAR characteristics are:

Purpose

Returns the virtual data address being accessed when a Watchpoint Debug Event was triggered.

Configuration

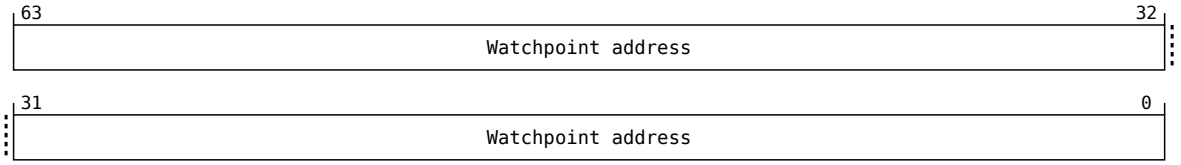
EDWAR is in the Core power domain

The value of this register is UNKNOWN if the PE is in Non-debug state, or if EDSCR.STATUS is not `0b101011`.

Attributes

EDWAR is a 64-bit register.

Field descriptions



ADDR, bits [63:0]

Watchpoint address. The data virtual address being accessed when a Watchpoint Debug Event was triggered and caused entry to Debug state. This address must be within a naturally-aligned block of memory of power-of-two size no larger than the DC ZVA block size.

The value of this register is UNKNOWN if the PE is in Non-debug state, or if Debug state was entered other than for a Watchpoint debug event.

The EDWAR is subject to the same alignment rules as the reporting of a watchpointed address in the FAR. See 'Exception syndrome information and preferred return address'.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing EDWAR

EDWAR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0x030`	EDWAR

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.2.42 MIDR_EL1, Main ID Register

The MIDR_EL1 characteristics are:

Purpose

Provides identification information for the PE, including an implementer code for the device and a device ID number.

Configuration

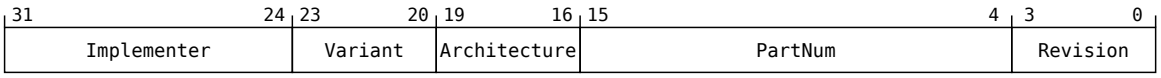
The power domain of MIDR_EL1 is IMPLEMENTATION DEFINED

External register MIDR_EL1 bits [31:0] are architecturally mapped to AArch64 System register MIDR_EL1[31:0].

Attributes

MIDR_EL1 is a 32-bit register.

Field descriptions



Implementer, bits [31:24]

The Implementer code. This field must hold an implementer code that has been assigned by Arm.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Implementer	Meaning
0x00	Reserved for software use.
0x41	Arm Limited.
0x42	Broadcom Corporation.
0x43	Cavium Inc.
0x44	Digital Equipment Corporation.
0x46	Fujitsu Ltd.
0x49	Infineon Technologies AG.
0x4D	Motorola or Freescale Semiconductor Inc.
0x4E	NVIDIA Corporation.
0x50	Applied Micro Circuits Corporation.
0x51	Qualcomm Inc.
0x56	Marvell International Ltd.
0x69	Intel Corporation.
0xC0	Ampere Computing.

Arm can assign codes that are not published in this manual. All values not assigned by Arm are reserved and must not be used.

Access to this field is RO.

Variant, bits [23:20]

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Variant number. Typically, this field is used to distinguish between different product variants, or major revisions of a product.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Architecture, bits [19:16]

Architecture version.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Architecture	Meaning
0b0001	Armv4.
0b0010	Armv4T.
0b0011	Armv5 (obsolete).
0b0100	Armv5T.
0b0101	Armv5TE.
0b0110	Armv5TEJ.
0b0111	Armv6.
0b1111	Architectural features are individually identified in the ID_* registers.

All other values are reserved.

Access to this field is RO.

PartNum, bits [15:4]

Primary Part Number for the device.

On processors implemented by Arm, if the top four bits of the primary part number are 0x0 or 0x7, the variant and architecture are encoded differently.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Revision, bits [3:0]

Revision number for the device.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing MIDR_EL1

MIDR_EL1 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0xD00	MIDR_EL1

Accessible as follows:

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register are IMPLEMENTATION DEFINED.
- Otherwise, accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.2.43 OSLAR_EL1, OS Lock Access Register

The OSLAR_EL1 characteristics are:

Purpose

Used to lock or unlock the OS Lock.

Configuration

OSLAR_EL1 is in the Core power domain

If FEAT_Debugv8p2 is not implemented, it is IMPLEMENTATION DEFINED whether external debug accesses to OSLAR_EL1 are ignored and return an error when AllowExternalDebugAccess() returns FALSE for the access.

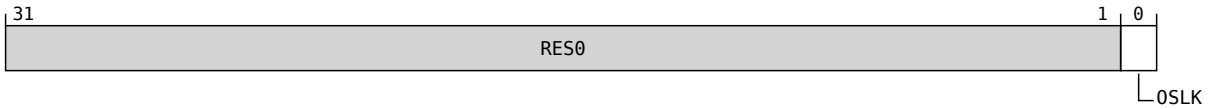
If FEAT_Debugv8p2 is implemented, external debug accesses to OSLAR_EL1 are ignored and return an error when AllowExternalDebugAccess() returns FALSE for the access.

External register OSLAR_EL1 bits [31:0] are architecturally mapped to AArch64 System register OSLAR_EL1[31:0].

Attributes

OSLAR_EL1 is a 32-bit register.

Field descriptions



Bits [31:1]

Reserved, RES0.

OSLK, bit [0]

On writes to OSLAR_EL1, bit[0] is copied to the OS Lock.

Use EDPRSR.OSLK to check the current status of the lock.

Accessing OSLAR_EL1

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalDebugAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

OSLAR_EL1 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0x300	OSLAR_EL1

Accessible as follows:

- When (DoubleLockStatus() || (!IsCorePowered())) || ((!AllowExternalDebugAccess(addrdesc)) && IsFeatureImplemented(FEAT_Debugv8p2)), accesses to this register return an ERROR.
- When AllowExternalDebugAccess(addrdesc) and SoftwareLockStatus(), accesses to this register are WI.
- When AllowExternalDebugAccess(addrdesc) and !SoftwareLockStatus(), accesses to this register are WO.
- Otherwise, accesses to this register are IMPLEMENTATION DEFINED.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.3 External trace registers

This section lists the external Trace unit registers.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.4 Cross-Trigger Interface registers

This section lists the Cross-Trigger Interface registers.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.4.1 ASICCTL, CTI External Multiplexer Control register

The ASICCTL characteristics are:

Purpose

Can be used to provide IMPLEMENTATION DEFINED controls for the CTI. For example, the register might be used to control multiplexors for additional IMPLEMENTATION DEFINED triggers. The IMPLEMENTATION DEFINED controls provided by this register might modify the architecturally defined behavior of the CTI.

Note

The architecturally-defined triggers must not be multiplexed.

Configuration

It is IMPLEMENTATION DEFINED whether ASICCTL is implemented in the Core power domain or in the Debug power domain

If it is implemented in the Core power domain, then it is IMPLEMENTATION DEFINED whether it is in the Cold reset domain or the Warm reset domain.

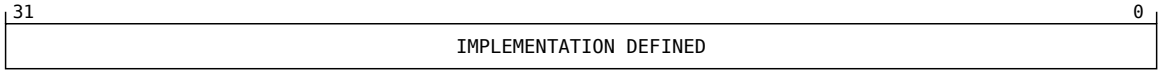
This register must reset to a value that supports the architecturally-defined behavior of the CTI. Changing the value of the register from its reset value causes IMPLEMENTATION DEFINED behavior that might differ from the architecturally-defined behavior of the CTI.

Other than the requirements listed in this register description, all aspects of the reset behavior of the ASICCTL are IMPLEMENTATION DEFINED.

Attributes

ASICCTL is a 32-bit register.

Field descriptions



IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

Accessing ASICCTL

ASICCTL can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0x144	ASICCTL

Accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalDebugAccess(addrdesc), and SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are IMPLEMENTATION DEFINED.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.4.2 CTIAPPCLEAR, CTI Application Trigger Clear register

The CTIAPPCLEAR characteristics are:

Purpose

Clears the application triggers.

Configuration

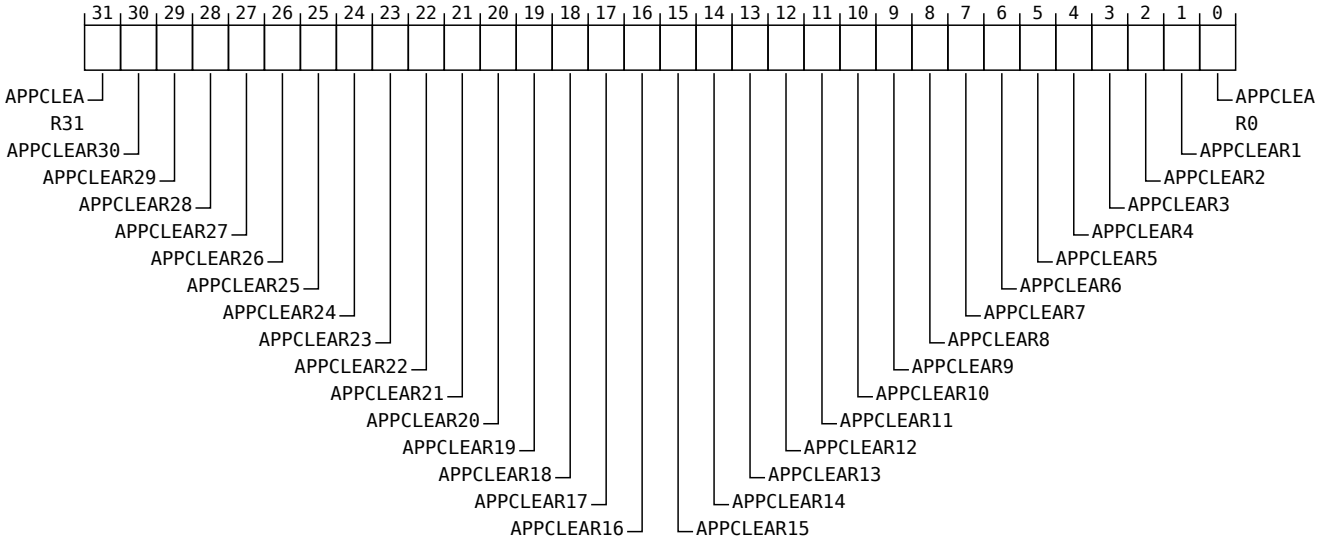
CTIAPPCLEAR is in the Debug power domain

External register CTIAPPCLEAR bits [31:0] are architecturally mapped to External register CTIAPPSET[31:0].

Attributes

CTIAPPCLEAR is a 32-bit register.

Field descriptions



APPCLEAR<x> , bits [x], for x = 31 to 0

Application trigger <x> disable.

Writing to this bit has the following effect:

APPCLEAR<x>	Meaning
0b0	No effect.
0b1	Clear corresponding application trigger to 0 and clear the corresponding channel event.

If the ECT does not support multicycle channel events, use of CTIAPPCLEAR is deprecated and the debugger must only use CTIAPPPULSE.

Accessing this field has the following behavior:

- When x >= UInt(CTIDEVID.NUMCHAN), access to this field is RAZ/WI.
- Otherwise, access to this field is RAZ/W1C.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing CTIAPPCLEAR

CTIAPPCLEAR can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0x018	CTIAPPCLEAR

Accessible as follows:

- When SoftwareLockStatus(), accesses to this register are WI.
- Otherwise, accesses to this register are WO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.4.3 CTIAPPPULSE, CTI Application Pulse register

The CTIAPPPULSE characteristics are:

Purpose

Causes event pulses to be generated on ECT channels.

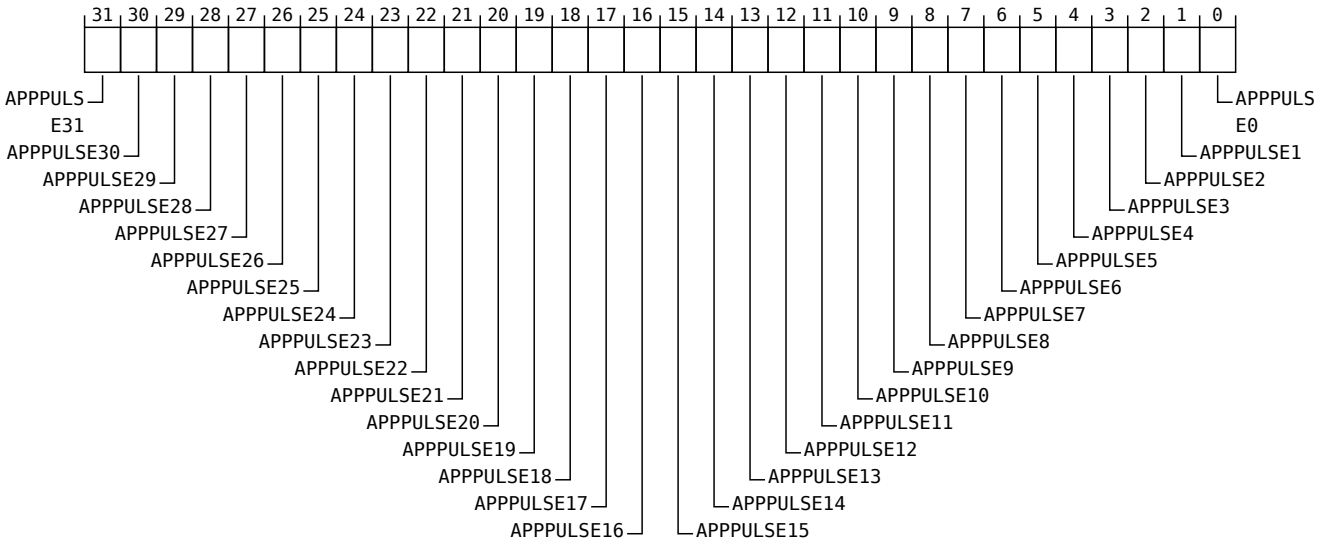
Configuration

CTIAPPPULSE is in the Debug power domain

Attributes

CTIAPPPULSE is a 32-bit register.

Field descriptions



APPPULSE<x> , bits [x], for x = 31 to 0

Generate event pulse on ECT channel <x>.

APPPULSE<x>	Meaning
0b0	No effect.
0b1	Channel <x> event pulse generated.

Note

- The CTIAPPPULSE operation does not affect the state of the application trigger. If the channel is active, either because of an earlier event or from the application trigger, then the value written to CTIAPPPULSE might have no effect.
- Multiple pulse events that occur close together might be merged into a single pulse event.

Accessing this field has the following behavior:

- When x >= UInt(CTIDEVID.NUMCHAN), access to this field is RAZ/WI.
- Otherwise, access to this field is WO/UNKNOWN.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing CTIAPPPULSE

It is CONSTRAINED UNPREDICTABLE whether a write to CTIAPPPULSE generates an event on a channel if CTICONTROL.GLBEN is 0.

CTIAPPPULSE can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0x01C	CTIAPPPULSE

Accessible as follows:

- When SoftwareLockStatus(), accesses to this register are WI.
- Otherwise, accesses to this register are WO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.4.4 CTIAPPSET, CTI Application Trigger Set register

The CTIAPPSET characteristics are:

Purpose

Sets the application triggers.

Configuration

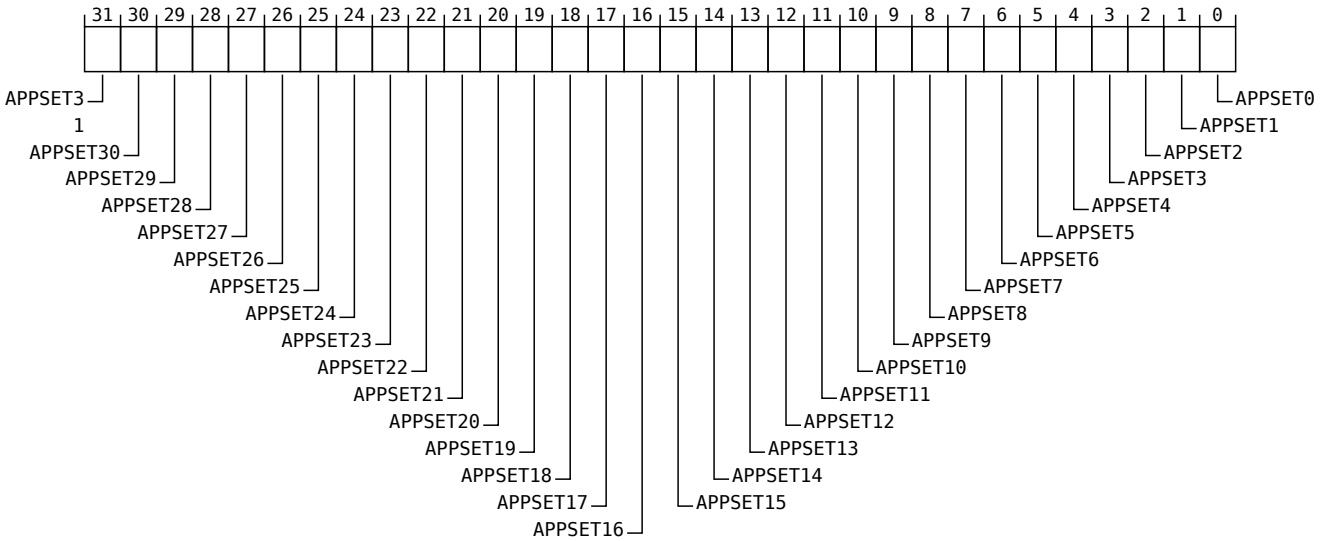
CTIAPPSET is in the Debug power domain

External register CTIAPPSET bits [31:0] are architecturally mapped to External register CTIAPPCLEAR[31:0].

Attributes

CTIAPPSET is a 32-bit register.

Field descriptions



APPSET<x> , bits [x], for x = 31 to 0

Application trigger <x> enable.

APPSET<x>	Meaning
0b0	Reading this means the application trigger is inactive. Writing this has no effect.
0b1	Reading this means the application trigger is active. Writing this sets the corresponding application trigger to 1 and generates a channel event.

If the ECT does not support multicycle channel events, use of CTIAPPSET is deprecated and the debugger must only use CTIAPPPULSE.

The reset behavior of this field is:

- On a External debug reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When x >= UInt(CTIDEVID.NUMCHAN), access to this field is RAZ/WI.
- Otherwise, access to this field is W1S.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing CTIAPPSET

CTIAPPSET can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0x014	CTIAPPSET

Accessible as follows:

- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.4.5 CTIAUTHSTATUS, CTI Authentication Status register

The CTIAUTHSTATUS characteristics are:

Purpose

Provides information about the state of the IMPLEMENTATION DEFINED authentication interface for CTI.

Configuration

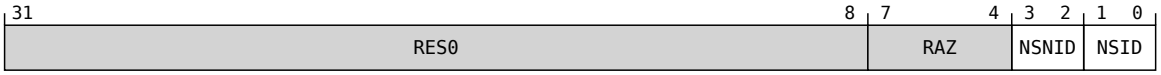
CTIAUTHSTATUS is in the Debug power domain

This register is OPTIONAL, and is required for CoreSight compliance.

Attributes

CTIAUTHSTATUS is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

Bits [7:4]

Reserved, RAZ.

NSNID, bits [3:2]

If EL3 is not implemented and the implemented Security state is Secure state, this field holds the same value as DBGAUTHSTATUS_EL1.SNID.

NSID, bits [1:0]

If EL3 is not implemented and the implemented Security state is Secure state, this field holds the same value as DBGAUTHSTATUS_EL1.SID.

Accessing CTIAUTHSTATUS

CTIAUTHSTATUS can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0xFB8	CTIAUTHSTATUS

Accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.4.6 CTICHINSTATUS, CTI Channel In Status register

The CTICHINSTATUS characteristics are:

Purpose

Provides the raw status of the ECT channel inputs to the CTI.

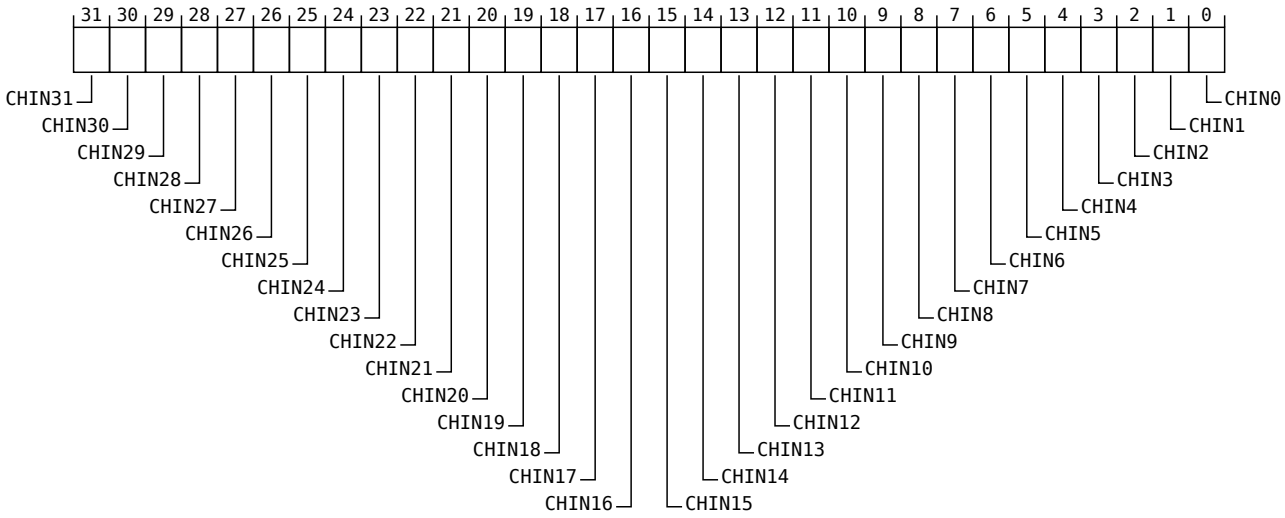
Configuration

CTICHINSTATUS is in the Debug power domain

Attributes

CTICHINSTATUS is a 32-bit register.

Field descriptions



CHIN<n> , bits [n], for n = 31 to 0

Input channel <n> status.

CHIN<n>	Meaning
0b0	Input channel <n> is inactive.
0b1	Input channel <n> is active.

If the ECT channels do not support multicycle events, then it is IMPLEMENTATION DEFINED whether an input channel can be observed as active.

Accessing this field has the following behavior:

- When n >= UInt(CTIDEVID.NUMCHAN), access to this field is RAZ/WI.
- Otherwise, access to this field is RO.

Accessing CTICHINSTATUS

CTICHINSTATUS can be accessed through the external debug interface:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Offset	Instance
CTI	0x138	CTICHINSTATUS

Accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.4.7 CTICHOUTSTATUS, CTI Channel Out Status register

The CTICHOUTSTATUS characteristics are:

Purpose

Provides the status of the ECT channel outputs from the CTI.

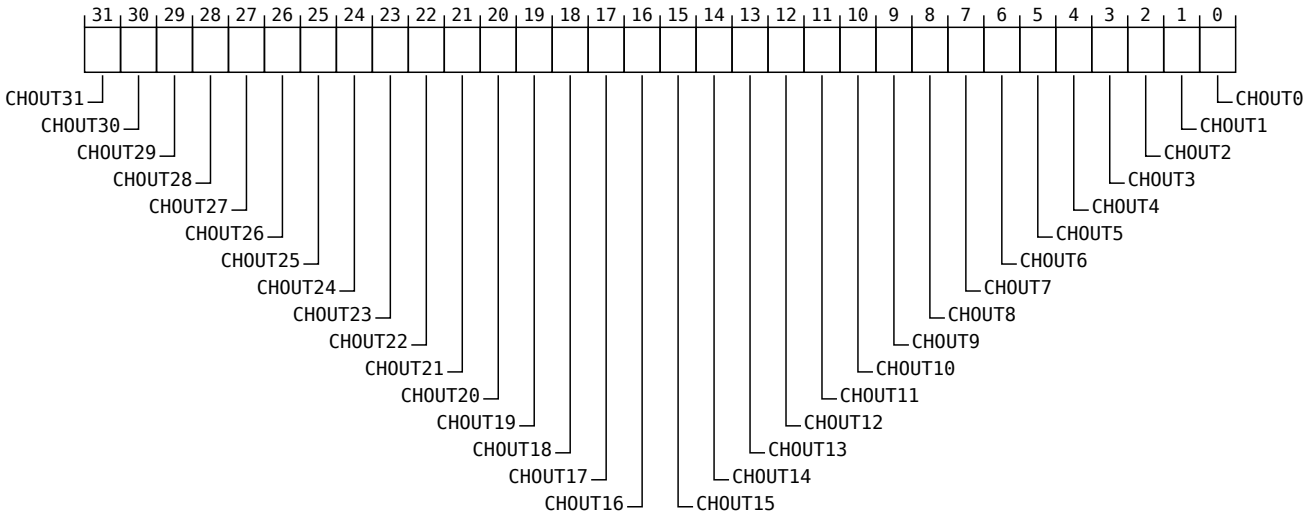
Configuration

CTICHOUTSTATUS is in the Debug power domain

Attributes

CTICHOUTSTATUS is a 32-bit register.

Field descriptions



CHOUT<n> , bits [n], for n = 31 to 0

Output channel <n> status.

CHOUT<n>	Meaning
0b0	Output channel <n> is inactive.
0b1	Output channel <n> is active.

If the ECT channels do not support multicycle events, then it is IMPLEMENTATION DEFINED whether an output channel can be observed as active.

Note

The value in CTICHOUTSTATUS is after gating by the channel gate. For more information, see CTIGATE.

Accessing this field has the following behavior:

- When n >= UInt(CTIDEVID.NUMCHAN), access to this field is RAZ/WI.
- Otherwise, access to this field is RO.

Accessing CTICHOUTSTATUS

CTICHOUTSTATUS can be accessed through the external debug interface:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Offset	Instance
CTI	0x13C	CTICHOUTSTATUS

Accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.4.8 CTICIDR0, CTI Component Identification Register 0

The CTICIDR0 characteristics are:

Purpose

Provides information to identify a CTI component.

For more information, see ‘About the Component Identification scheme’.

Configuration

CTICIDR0 is in the Debug power domain

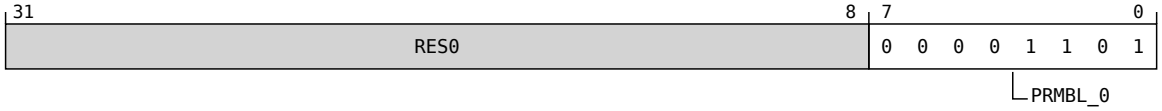
This register is required for CoreSight compliance.

Implementation of this register is OPTIONAL.

Attributes

CTICIDR0 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

PRMBL_0, bits [7:0]

Preamble.

Reads as `0x0D`

Access to this field is RO.

Accessing CTICIDR0

CTICIDR0 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFF0`	CTICIDR0

Accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.4.9 CTICIDR1, CTI Component Identification Register 1

The CTICIDR1 characteristics are:

Purpose

Provides information to identify a CTI component.

For more information, see ‘About the Component Identification scheme’.

Configuration

CTICIDR1 is in the Debug power domain

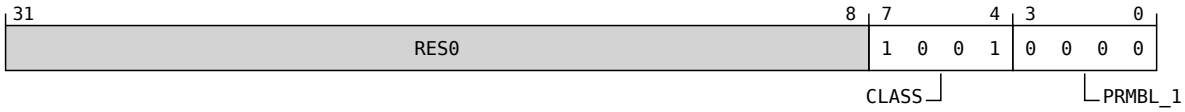
This register is required for CoreSight compliance.

Implementation of this register is OPTIONAL.

Attributes

CTICIDR1 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

CLASS, bits [7:4]

Component class.

CLASS	Meaning
0b1001	CoreSight component.

Other values are defined by the CoreSight Architecture.

This field reads as 0x9.

Access to this field is RO.

PRMBL_1, bits [3:0]

Preamble.

Reads as 0b0000

Access to this field is RO.

Accessing CTICIDR1

CTICIDR1 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0xFF4	CTICIDR1

Accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.4.10 CTICIDR2, CTI Component Identification Register 2

The CTICIDR2 characteristics are:

Purpose

Provides information to identify a CTI component.

For more information, see ‘About the Component Identification scheme’.

Configuration

CTICIDR2 is in the Debug power domain

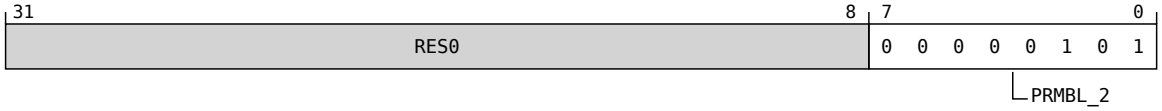
This register is required for CoreSight compliance.

Implementation of this register is OPTIONAL.

Attributes

CTICIDR2 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

PRMBL_2, bits [7:0]

Preamble.

Reads as `0x05`

Access to this field is RO.

Accessing CTICIDR2

CTICIDR2 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFF8`	CTICIDR2

Accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.4.11 CTICIDR3, CTI Component Identification Register 3

The CTICIDR3 characteristics are:

Purpose

Provides information to identify a CTI component.

For more information, see ‘About the Component Identification scheme’.

Configuration

CTICIDR3 is in the Debug power domain

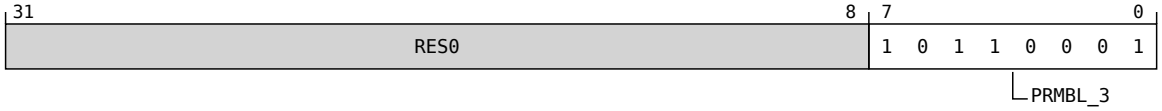
This register is required for CoreSight compliance.

Implementation of this register is OPTIONAL.

Attributes

CTICIDR3 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

PRMBL_3, bits [7:0]

Preamble.

Reads as `0xB1`

Access to this field is RO.

Accessing CTICIDR3

CTICIDR3 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFFC`	CTICIDR3

Accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.4.12 CTICLAIMCLR, CTI CLAIM Tag Clear register

The CTICLAIMCLR characteristics are:

Purpose

Used by software to read the values of the CLAIM bits, and to clear CLAIM tag bits to 0.

Configuration

CTICLAIMCLR is in the Debug power domain

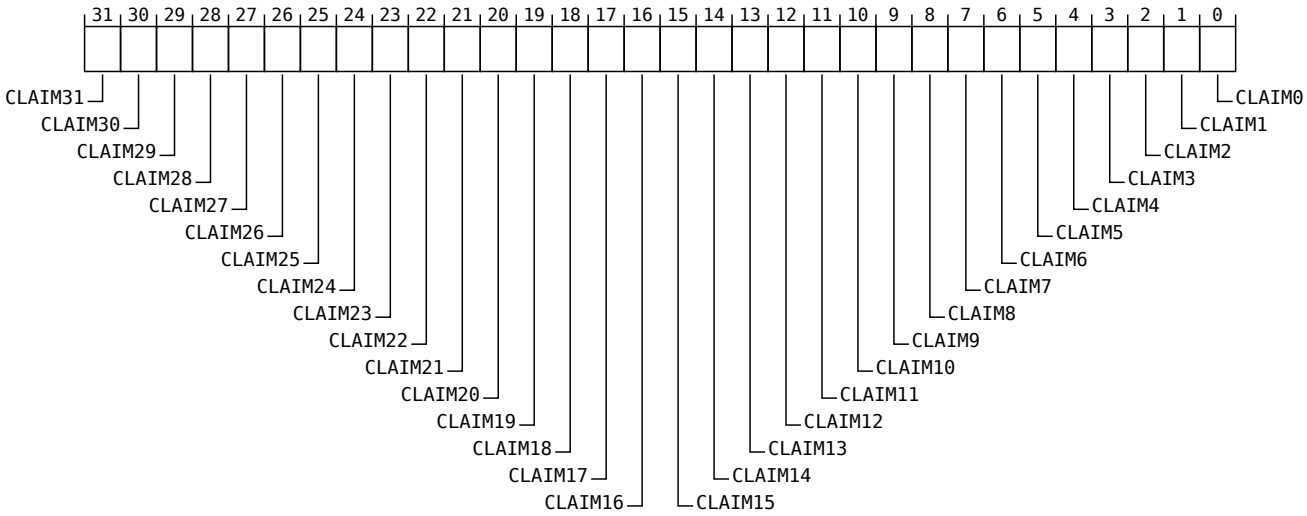
External register CTICLAIMCLR bits [31:0] are architecturally mapped to External register CTICLAIMSET[31:0].

Implementation of this register is OPTIONAL.

Attributes

CTICLAIMCLR is a 32-bit register.

Field descriptions



CLAIM<m> , bits [m], for m = 31 to 0

Claim Tag Clear. Indicates the current status of Claim Tag bit <m>, and is used to clear Claim Tag bit <m> to 0.

CLAIM<m>	Meaning
0b0	On a read: Claim Tag bit <m> is not set. On a write: Ignored.
0b1	On a read: Claim Tag bit <m> is set. On a write: Clear Claim tag bit <m> to 0.

The number of Claim Tag bits implemented is indicated in CTICLAIMSET.

An External Debug reset clears the CLAIM tag bits to 0.

Accessing this field has the following behavior:

- When m >= NUM_CTI_CLAIM_TAGS, access to this field is RAZ/WI.
- Otherwise, access to this field is W1C.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing CTICLAIMCLR

CTICLAIMCLR can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0xFA4	CTICLAIMCLR

Accessible as follows:

- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.4.13 CTICLAIMSET, CTI CLAIM Tag Set register

The CTICLAIMSET characteristics are:

Purpose

Used by software to set CLAIM bits to 1.

Configuration

CTICLAIMSET is in the Debug power domain

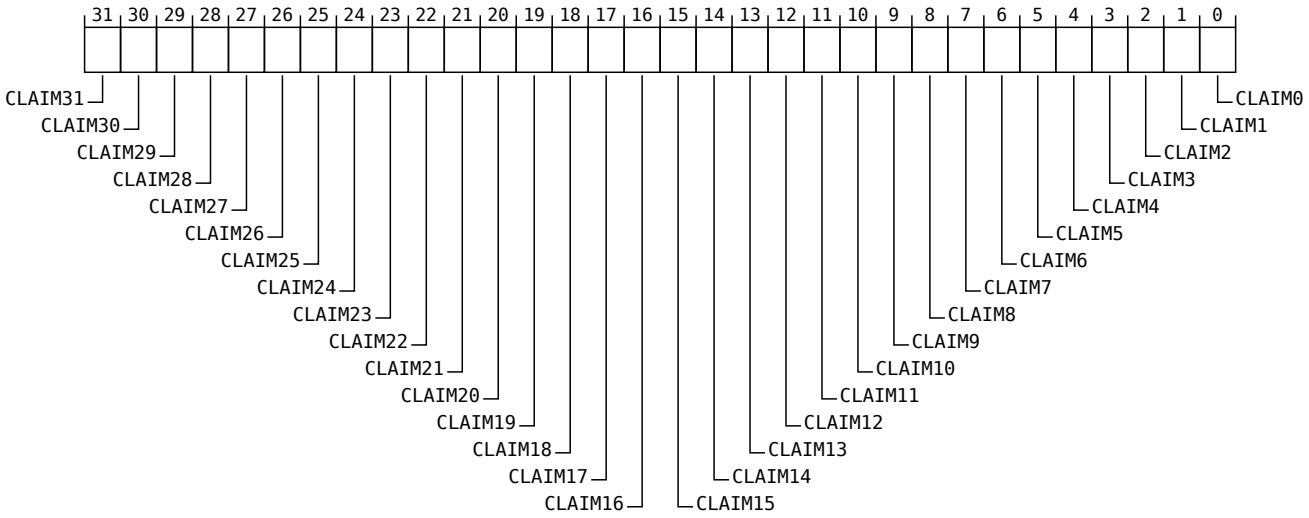
External register CTICLAIMSET bits [31:0] are architecturally mapped to External register CTICLAIMCLR[31:0].

Implementation of this register is OPTIONAL.

Attributes

CTICLAIMSET is a 32-bit register.

Field descriptions



CLAIM<m> , bits [m], for m = 31 to 0

Claim Tag Set. Indicates whether Claim Tag bit <m> is implemented, and is used to set Claim Tag bit <m> to 1.

CLAIM<m>	Meaning
0b0	On a read: Claim Tag bit <m> is not implemented. On a write: Ignored.
0b1	On a read: Claim Tag bit <m> is implemented. On a write: Set Claim Tag bit <m> to 1.

An External Debug reset clears the CLAIM tag bits to 0.

Accessing this field has the following behavior:

- When m >= NUM_CTI_CLAIM_TAGS, access to this field is RAZ/WI.
- Otherwise, access to this field is RAO/W1S.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing CTICLAIMSET

CTICLAIMSET can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0xFA0	CTICLAIMSET

Accessible as follows:

- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.4.14 CTICONTROL, CTI Control register

The CTICONTROL characteristics are:

Purpose

Controls whether the CTI is enabled.

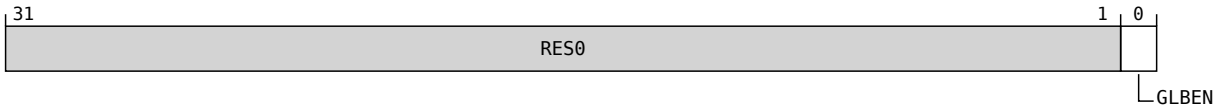
Configuration

CTICONTROL is in the Debug power domain

Attributes

CTICONTROL is a 32-bit register.

Field descriptions



Bits [31:1]

Reserved, RES0.

GLBEN, bit [0]

Enables or disables the CTI mapping functions. Possible values of this field are:

GLBEN	Meaning
0b0	CTI mapping functions and application trigger disabled.
0b1	CTI mapping functions and application trigger enabled.

When GLBEN is 0, the input channel to output trigger, input trigger to output channel, and application trigger functions are disabled and do not signal new events on either output triggers or output channels. If a previously asserted output trigger has not been acknowledged, it is CONSTRAINED UNPREDICTABLE which of the following occurs:

- The output trigger remains asserted after the mapping functions are disabled.
- The output trigger is deasserted after the mapping functions are disabled.

All output triggers are disabled by CTI reset.

If the ECT supports multicycle channel events any existing output channel events will be terminated.

The reset behavior of this field is:

- On a External debug reset, this field resets to '0'.

Accessing CTICONTROL

CTICONTROL can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0x000	CTICONTROL

Accessible as follows:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.4.15 CTIDEVAFF0, CTI Device Affinity register 0

The CTIDEVAFF0 characteristics are:

Purpose

Copy of the low half of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the CTI component relates to.

Configuration

CTIDEVAFF0 is in the Debug power domain

If the CTI is CTIv1, this register is OPTIONAL. If the CTI is CTIv2, this register is mandatory.

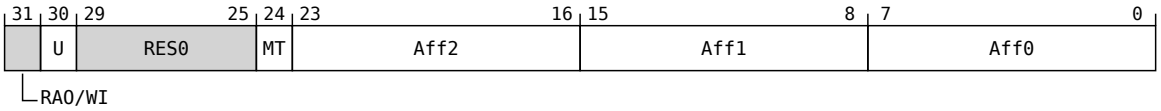
Arm recommends that the CTI is CTIv2.

If this register is implemented, then CTIDEVAFF1 must also be implemented. If the CTI of a PE does not implement the CTI Device Affinity registers, the CTI block of the external debug memory map must be located 64KB above the debug registers in the external debug interface.

Attributes

CTIDEVAFF0 is a 32-bit register.

Field descriptions



Bit [31]

Reserved, RAO/WI.

U, bit [30]

Indicates a Uniprocessor system, as distinct from PE 0 in a multiprocessor system.

The value of this field is an IMPLEMENTATION DEFINED choice of:

U	Meaning
0b0	Processor is part of a multiprocessor system.
0b1	Processor is part of a uniprocessor system.

Access to this field is RO.

Bits [29:25]

Reserved, RES0.

MT, bit [24]

Indicates whether the lowest level of affinity consists of logical PEs that are implemented using an interdependent approach, such as multithreading. See the description of Aff0 for more information about affinity levels.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MT	Meaning
0b0	Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is largely independent.
0b1	Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is very interdependent.

Note

This field does not indicate that multithreading is implemented and does not indicate that PEs with different affinity level 0 values, and the same values for affinity level 1 and higher are implemented.

Access to this field is RO.

Aff2, bits [23:16]

Affinity level 2. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Aff1, bits [15:8]

Affinity level 1. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Aff0, bits [7:0]

Affinity level 0. The value of the MPIDR.{Aff2, Aff1, Aff0} or MPIDR_EL1.{Aff3, Aff2, Aff1, Aff0} set of fields of each PE must be unique within the system as a whole.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing CTIDEVAFF0

CTIDEVAFF0 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0xFA8	CTIDEVAFF0

Accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.4.16 CTIDEVAFF1, CTI Device Affinity register 1

The CTIDEVAFF1 characteristics are:

Purpose

Copy of the high half of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the CTI component relates to.

Configuration

CTIDEVAFF1 is in the Debug power domain

If the CTI is CTIv1, this register is OPTIONAL. If the CTI is CTIv2, this register is mandatory.

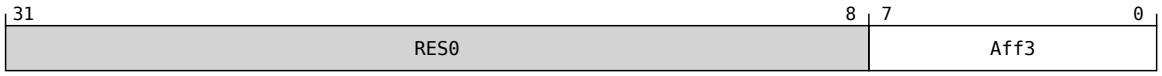
Arm recommends that the CTI is CTIv2.

If this register is implemented, then CTIDEVAFF0 must also be implemented. If the CTI of a PE does not implement the CTI Device Affinity registers, the CTI block of the external debug memory map must be located 64KB above the debug registers in the external debug interface.

Attributes

CTIDEVAFF1 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

Aff3, bits [7:0]

Affinity level 3. See the description of CTIDEVAFF0.Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing CTIDEVAFF1

CTIDEVAFF1 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0xFAC	CTIDEVAFF1

Accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.4.17 CTIDEVARCH, CTI Device Architecture register

The CTIDEVARCH characteristics are:

Purpose

Identifies the programmers' model architecture of the CTI component.

Configuration

CTIDEVARCH is in the Debug power domain

If the CTI is CTIv1, this register is OPTIONAL. If the CTI is CTIv2, this register is mandatory.

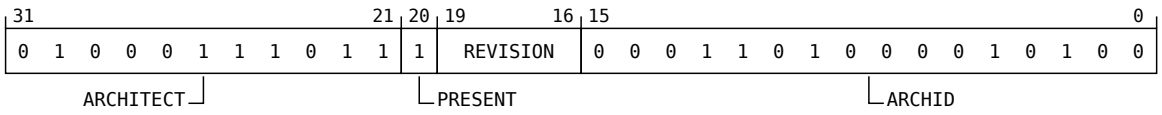
Arm recommends that the CTI is CTIv2.

If this register is not implemented, CTIDEVAFF0 and CTIDEVAFF1 are also not implemented.

Attributes

CTIDEVARCH is a 32-bit register.

Field descriptions



ARCHITECT, bits [31:21]

Defines the architect of the component. For CTI, this is Arm Limited.

Bits [31:28] are the JEP106 continuation code, `0b0100`.

Bits [27:21] are the JEP106 identification code, `0b0111011`.

Reads as `0b01000111011`

Access to this field is RO.

PRESENT, bit [20]

DEVARCH present. Indicates that the CTIDEVARCH register is present.

Reads as `0b1`

Access to this field is RO.

REVISION, bits [19:16]

Revision. Defines the architecture revision of the component.

The value of this field is an IMPLEMENTATION DEFINED choice of:

REVISION	Meaning
`0b0000`	First revision.
`0b0001`	As `0b0000`, and also adds support for CTIDEVCTL.

All other values are reserved.

When FEAT_DoPD is implemented, the value `0b0000` is not permitted.

Access to this field is RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ARCHID, bits [15:0]

Defines this part to be an Armv8 debug component. For architectures defined by Arm this is further subdivided.

For CTI:

- Bits [15:12] are the architecture version, `0x1`.
- Bits [11:0] are the architecture part number, `0xA14`.

This corresponds to CTI architecture version CTIv2.

Reads as `0x1A14`

Access to this field is RO.

Accessing CTIDEVARCH

CTIDEVARCH can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFBC`	CTIDEVARCH

Accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.4.18 CTIDEVCTL, CTI Device Control register

The CTIDEVCTL characteristics are:

Purpose

Provides target-specific device controls

Configuration

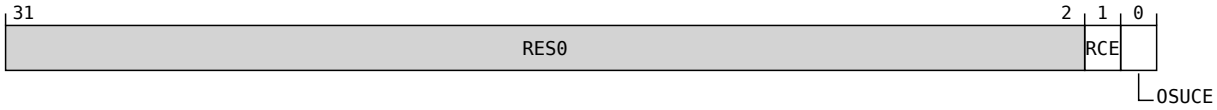
CTIDEVCTL is in the Debug power domain

This register is present only when FEAT_DoPD is implemented. Otherwise, direct accesses to CTIDEVCTL are RES0.

Attributes

CTIDEVCTL is a 32-bit register.

Field descriptions



Bits [31:2]

Reserved, RES0.

RCE, bit [1]

Reset Catch Enable.

RCE	Meaning
0b0	Reset Catch debug event disabled.
0b1	Reset Catch debug event enabled.

The reset behavior of this field is:

- On a External debug reset, this field resets to '0'.

OSUCE, bit [0]

OS Unlock Catch Enable

OSUCE	Meaning
0b0	OS Unlock Catch debug event disabled.
0b1	OS Unlock Catch debug event enabled.

The reset behavior of this field is:

- On a External debug reset, this field resets to '0'.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing CTIDEVCTL

CTIDEVCTL can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0x150	CTIDEVCTL

Accessible as follows:

- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.4.19 CTIDEVID, CTI Device ID register 0

The CTIDEVID characteristics are:

Purpose

Describes the CTI component to the debugger.

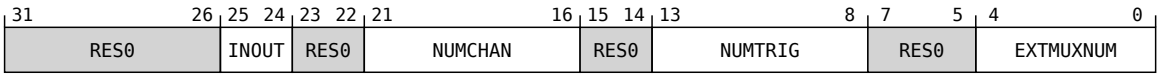
Configuration

CTIDEVID is in the Debug power domain

Attributes

CTIDEVID is a 32-bit register.

Field descriptions



Bits [31:26]

Reserved, RES0.

INOUT, bits [25:24]

Input/output options. Indicates presence of the input gate. If the CTM is not implemented or CTIv2 is not implemented, this field is RAZ.

The value of this field is an IMPLEMENTATION DEFINED choice of:

INOUT	Meaning
0b00	CTIGATE does not mask propagation of input events from external channels.
0b01	CTIGATE masks propagation of input events from external channels.

All other values are reserved.

Access to this field is RO.

Bits [23:22]

Reserved, RES0.

NUMCHAN, bits [21:16]

Number of ECT channels implemented. For Armv8, valid values are:

- 0b000011 3 channels (0..2) implemented.
- 0b000100 4 channels (0..3) implemented.
- 0b000101 5 channels (0..4) implemented.
- 0b000110 6 channels (0..5) implemented.

This field has an IMPLEMENTATION DEFINED value.

and so on up to 0b100000, 32 channels (0..31) implemented.

All other values are reserved.

Access to this field is RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [15:14]

Reserved, RES0.

NUMTRIG, bits [13:8]

Upper bound for number of triggers. The indices of all implemented input and output triggers are less than this value.

This field has an IMPLEMENTATION DEFINED value.

There is no guarantee that all of the input and output triggers, including the highest numbered, are connected to any components, or that the implementation of input and output triggers is symmetrical.

Access to this field is RO.

Bits [7:5]

Reserved, RES0.

EXTMUXNUM, bits [4:0]

Number of multiplexors available on triggers. This value is used in conjunction with External Control register, ASICCTL.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing CTIDEVID

CTIDEVID can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0xFC8	CTIDEVID

Accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.4.20 CTIDEVID1, CTI Device ID register 1

The CTIDEVID1 characteristics are:

Purpose

Reserved for future information about the CTI component to the debugger.

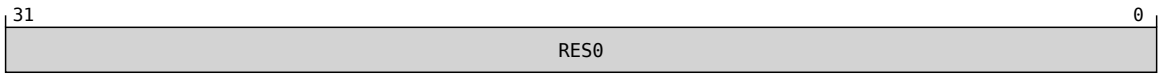
Configuration

CTIDEVID1 is in the Debug power domain

Attributes

CTIDEVID1 is a 32-bit register.

Field descriptions



Bits [31:0]

Reserved, RES0.

Accessing CTIDEVID1

CTIDEVID1 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0xFC4	CTIDEVID1

Accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.4.21 CTIDEVID2, CTI Device ID register 2

The CTIDEVID2 characteristics are:

Purpose

Reserved for future information about the CTI component to the debugger.

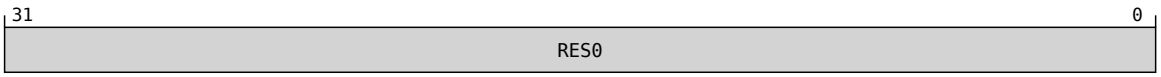
Configuration

CTIDEVID2 is in the Debug power domain

Attributes

CTIDEVID2 is a 32-bit register.

Field descriptions



Bits [31:0]

Reserved, RES0.

Accessing CTIDEVID2

CTIDEVID2 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0xFC0	CTIDEVID2

Accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.4.22 CTIDEVTYPE, CTI Device Type register

The CTIDEVTYPE characteristics are:

Purpose

Indicates to a debugger that this component is part of a PE's cross-trigger interface.

Configuration

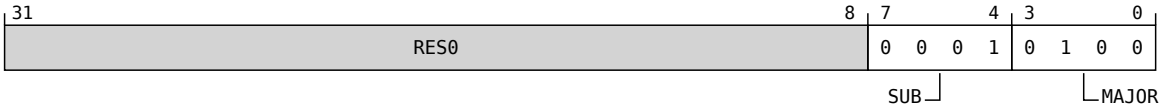
CTIDEVTYPE is in the Debug power domain

Implementation of this register is OPTIONAL.

Attributes

CTIDEVTYPE is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

SUB, bits [7:4]

Subtype. Indicates this is a component within a PE.

Reads as `0b0001`

Access to this field is RO.

MAJOR, bits [3:0]

Major type. Indicates this is a cross-trigger component.

Reads as `0b0100`

Access to this field is RO.

Accessing CTIDEVTYPE

CTIDEVTYPE can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFCC`	CTIDEVTYPE

Accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.4.23 CTIGATE, CTI Channel Gate Enable register

The CTIGATE characteristics are:

Purpose

Determines whether events on channels propagate through the CTM to other ECT components, or from the CTM into the CTI.

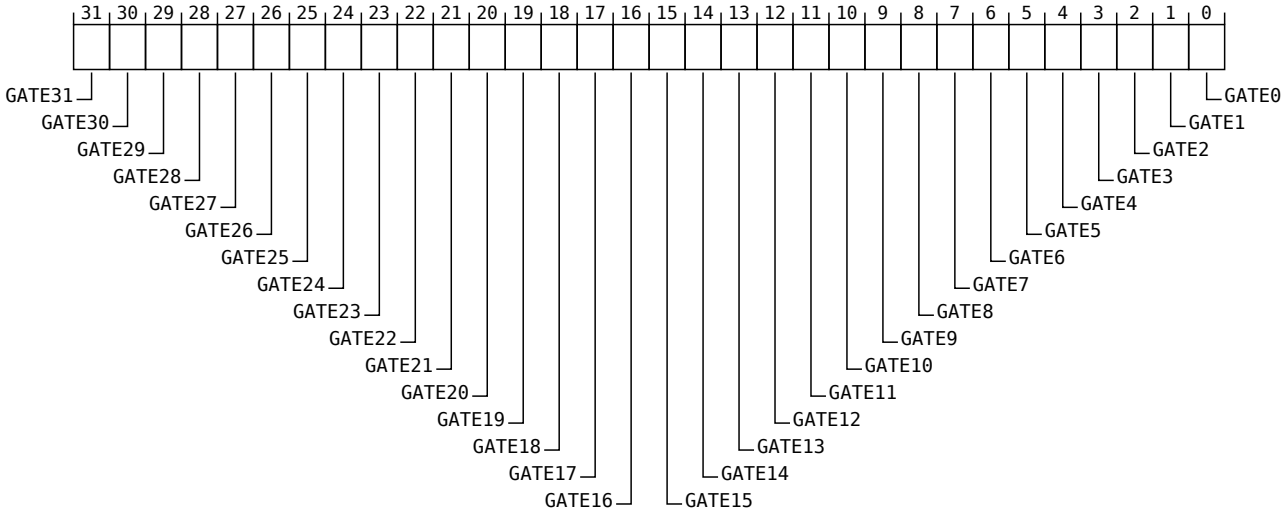
Configuration

CTIGATE is in the Debug power domain

Attributes

CTIGATE is a 32-bit register.

Field descriptions



GATE<x> , bits [x], for x = 31 to 0

Channel <x> gate enable.

GATE<x>	Meaning
0b0	Disable output and, if CTIDEVID.INOUT == 0b01, input channel <x> propagation.
0b1	Enable output and, if CTIDEVID.INOUT == 0b01, input channel <x> propagation.

If GATE<x> is set to 0, no new events will be propagated to the ECT, and if the ECT supports multicycle channel events any existing output channel events will be terminated.

The reset behavior of this field is:

- On a External debug reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When x >= UInt(CTIDEVID.NUMCHAN), access to this field is RAZ/WI.
- Otherwise, access to this field is RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing CTIGATE

CTIGATE can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0x140	CTIGATE

Accessible as follows:

- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.4.24 CTIINEN<n>, CTI Input Trigger to Output Channel Enable registers, n = 0 - 31

The CTIINEN<n> characteristics are:

Purpose

Enables the signaling of an event on output channels when input trigger event n is received by the CTI.

Configuration

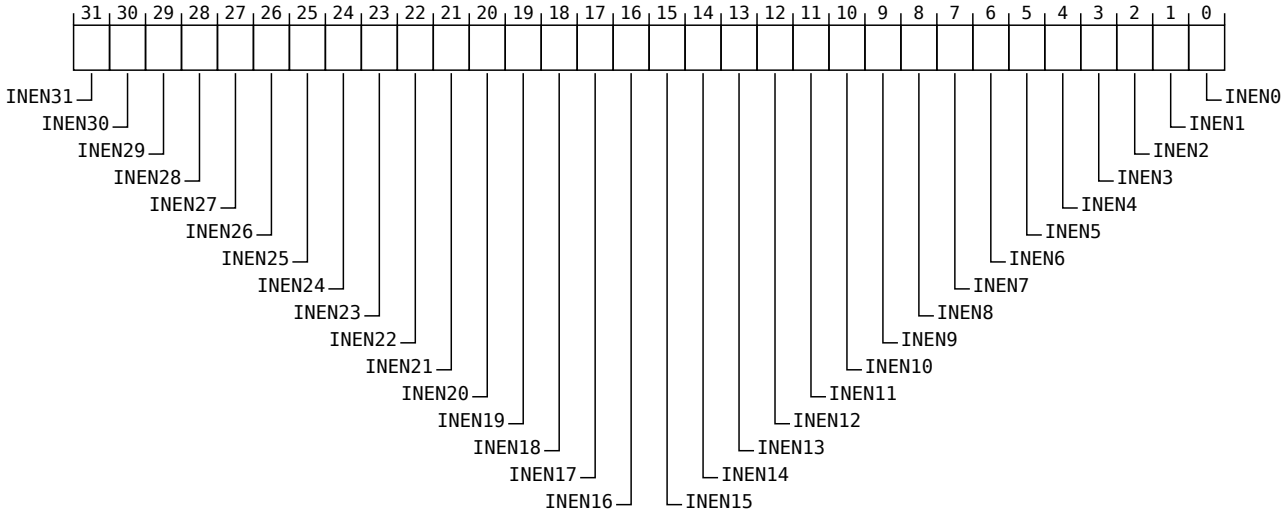
CTIINEN<n> is in the Debug power domain

If input trigger n is not implemented or not connected, CTIINEN<n> is RES0.

Attributes

CTIINEN<n> is a 32-bit register.

Field descriptions



INEN<x> , bits [x], for x = 31 to 0

Input trigger <n> to output channel <x> enable.

INEN<x>	Meaning
0b0	Input trigger <n> will not generate an event on output channel <x>.
0b1	Input trigger <n> will generate an event on output channel <x>.

The reset behavior of this field is:

- On a External debug reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When x >= UInt(CTIDEVID.NUMCHAN), access to this field is RAZ/WI.
- Otherwise, access to this field is RW.

Accessing CTIINEN<n>

CTIINEN<n> can be accessed through the external debug interface:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Offset	Instance
CTI	0x020 + (4 * ↪n)	CTIINEN<n>

Accessible as follows:

- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.4.25 CTIINTACK, CTI Output Trigger Acknowledge register

The CTIINTACK characteristics are:

Purpose

Can be used to deactivate the output triggers.

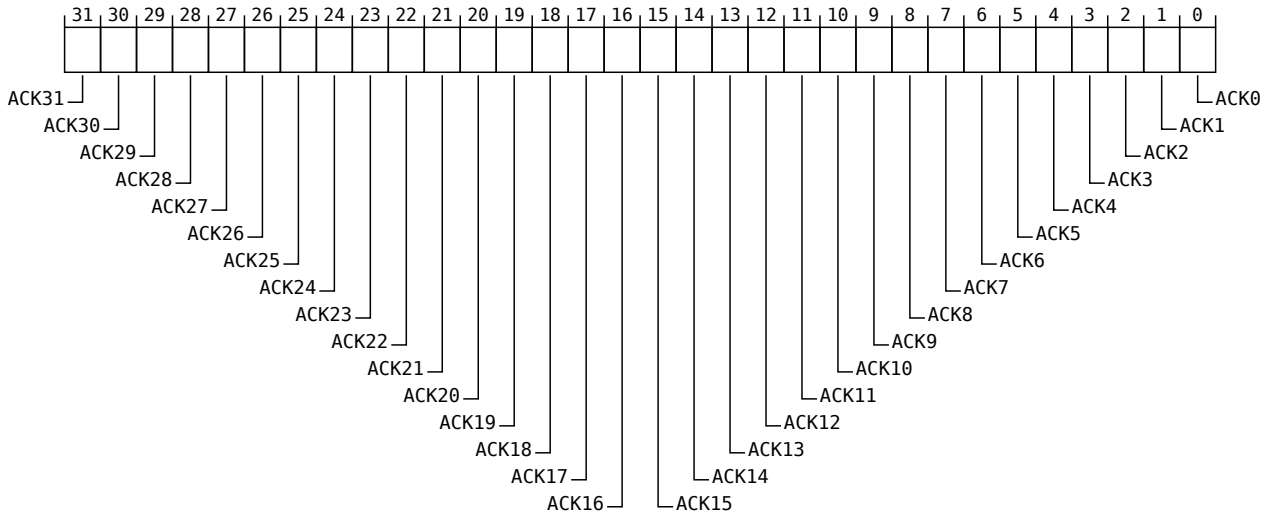
Configuration

CTIINTACK is in the Debug power domain

Attributes

CTIINTACK is a 32-bit register.

Field descriptions



ACK<n> , bits [n], for n = 31 to 0

Acknowledge for output trigger <n>.

If any of the following is true, writes to ACK<n> are ignored:

- n >= CTIDEVID.NUMTRIG, the number of implemented triggers.
- Output trigger n is not active.
- The channel mapping function output, as controlled by CTIOUTEN<n>, is still active.

Otherwise, if any of the following are true, ACK<n> is RES0:

- Output trigger n is not implemented.
- Output trigger n is not connected.
- Output trigger n is self-acknowledging and does not require software acknowledge.

Otherwise, the behavior on writes to ACK<n> is as follows:

ACK<n>	Meaning
0b0	No effect
0b1	Deactivate the trigger.

Accessing this field has the following behavior:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When n >= UInt(CTIDEVID.NUMTRIG), access to this field is RAZ/WI.
- Otherwise, access to this field is WO/UNKNOWN.

Accessing CTIINTACK

A debugger must read CTITRIGOUTSTATUS to confirm that the output trigger has been acknowledged before generating any event that must be ordered after the write to CTIINTACK, such as a write to CTIAPPPULSE to activate another trigger.

CTIINTACK can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0x010`	CTIINTACK

Accessible as follows:

- When SoftwareLockStatus(), accesses to this register are WI.
- Otherwise, accesses to this register are WO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.4.26 CTIITCTRL, CTI Integration mode Control register

The CTIITCTRL characteristics are:

Purpose

Enables the CTI to switch from its default mode into integration mode, where test software can control directly the inputs and outputs of the PE, for integration testing or topology detection.

Configuration

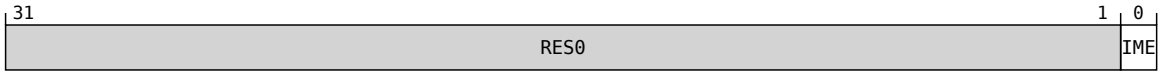
The power domain of CTIITCTRL is IMPLEMENTATION DEFINED

Implementation of this register is OPTIONAL.

Attributes

CTIITCTRL is a 32-bit register.

Field descriptions



Bits [31:1]

Reserved, RES0.

IME, bit [0]

Integration mode enable. When IME == 1, the device reverts to an integration mode to enable integration testing or topology detection.

IME	Meaning
0b0	Normal operation.
0b1	Integration mode enabled.

The integration mode behavior is IMPLEMENTATION DEFINED.

The following resets apply:

- If the register is implemented in the Core power domain:
  - On a Cold reset, this field resets to 0.
  - On an External debug reset, the value of this field is unchanged.
  - On a Warm reset, the value of this field is unchanged.
- If the register is implemented in the External debug power domain:
  - On a Cold reset, the value of this field is unchanged.
  - On an External debug reset, this field resets to 0.
  - On a Warm reset, the value of this field is unchanged.

Accessing CTIITCTRL

CTIITCTRL can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0xF00	CTIITCTRL

Accessible as follows:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), accesses to this register are IMPLEMENTATION DEFINED.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.4.27 CTILAR, CTI Lock Access Register

The CTILAR characteristics are:

Purpose

Allows or disallows access to the CTI registers through a memory-mapped interface.

The optional Software Lock provides a lock to prevent memory-mapped writes to the Cross-Trigger Interface registers. Use of this lock mechanism reduces the risk of accidental damage to the contents of the Cross-Trigger Interface registers. It does not, and cannot, prevent all accidental or malicious damage.

Configuration

CTILAR is in the Debug power domain

If FEAT_Debugv8p4 is implemented, the Software Lock is not implemented.

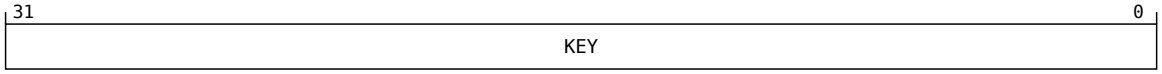
Software uses CTILAR to set or clear the lock, and CTILSR to check the current status of the lock.

Attributes

CTILAR is a 32-bit register.

Field descriptions

When CTI Software Lock is implemented:

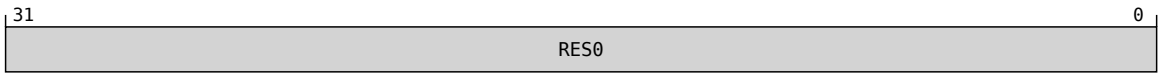


KEY, bits [31:0]

Lock Access control. Writing the key value `0xC5ACCE55` to this field unlocks the lock, enabling write accesses to this component's registers through a memory-mapped interface.

Writing any other value to this register locks the lock, disabling write accesses to this component's registers through a memory mapped interface.

Otherwise:



Otherwise

Bits [31:0]

Reserved, RES0.

Accessing CTILAR

CTILAR can be accessed through the memory-mapped interface:

Component	Offset	Instance
CTI	`0xFB0`	CTILAR

Accesses to this register are WO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.4.28 CTILSR, CTI Lock Status Register

The CTILSR characteristics are:

Purpose

Indicates the current status of the Software Lock for CTI registers.

The optional Software Lock provides a lock to prevent memory-mapped writes to the Cross-Trigger Interface registers. Use of this lock mechanism reduces the risk of accidental damage to the contents of the Cross-Trigger Interface registers. It does not, and cannot, prevent all accidental or malicious damage.

Configuration

CTILSR is in the Debug power domain

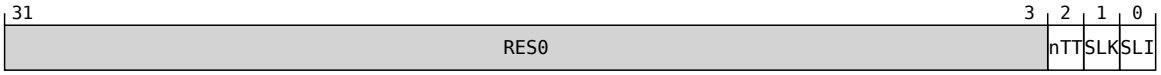
If FEAT_Debugv8p4 is implemented, the Software Lock is not implemented.

Software uses CTILAR to set or clear the lock, and CTILSR to check the current status of the lock.

Attributes

CTILSR is a 32-bit register.

Field descriptions



Bits [31:3]

Reserved, RES0.

nTT, bit [2]

Not thirty-two bit access required. RAZ.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

SLK, bit [1]

When CTI Software Lock is implemented:

Software Lock status for this component. For an access to LSR that is not a memory-mapped access, or when the Software Lock is not implemented, this field is RES0.

For memory-mapped accesses when the Software Lock is implemented, possible values of this field are:

SLK	Meaning
0b0	Lock clear. Writes are permitted to this component's registers.
0b1	Lock set. Writes to this component's registers are ignored, and reads have no side effects.

The reset behavior of this field is:

- On a External debug reset, this field resets to '1'.

Otherwise:

Reserved, RAZ.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SLI, bit [0]

Software Lock implemented. For an access to LSR that is not a memory-mapped access, this field is RAZ.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SLI	Meaning
0b0	Software Lock not implemented or not memory-mapped access.
0b1	Software Lock implemented and memory-mapped access.

Access to this field is RO.

Accessing CTILSR

CTILSR can be accessed through the memory-mapped interface:

Component	Offset	Instance
CTI	0xFB4	CTILSR

Accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.4.29 CTIOUTEN<n>, CTI Input Channel to Output Trigger Enable registers, n = 0 - 31

The CTIOUTEN<n> characteristics are:

Purpose

Defines which input channels generate output trigger n.

Configuration

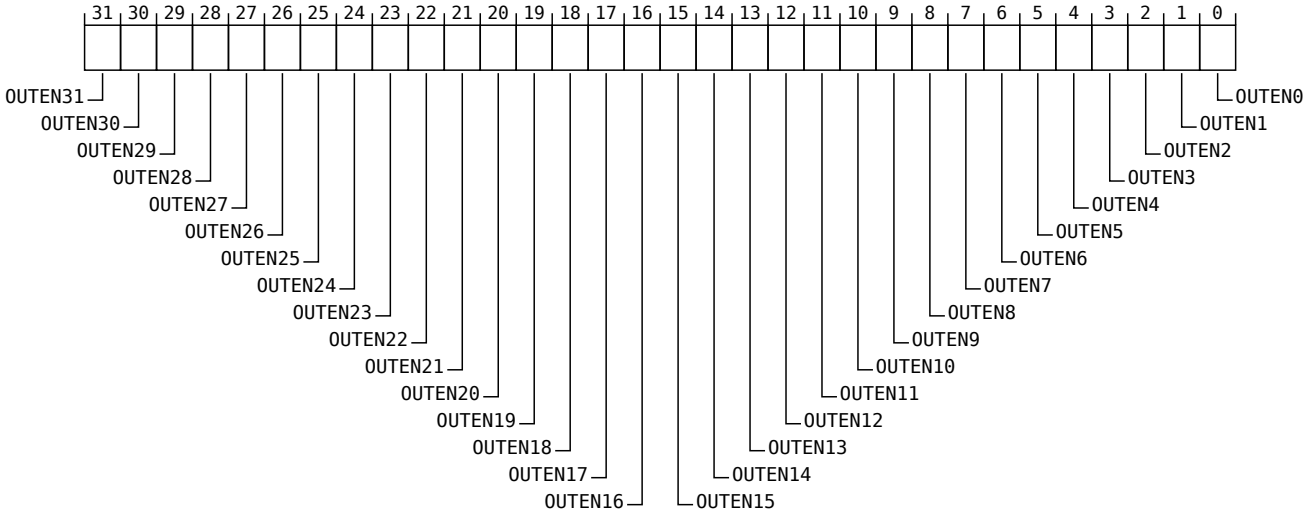
CTIOUTEN<n> is in the Debug power domain

If output trigger n is not implemented or not connected, CTIOUTEN<n> is RES0.

Attributes

CTIOUTEN<n> is a 32-bit register.

Field descriptions



OUTEN<x> , bits [x], for x = 31 to 0

Input channel <x> to output trigger <n> enable.

OUTEN<x>	Meaning
0b0	An event on input channel <x> will not cause output trigger <n> to be asserted.
0b1	An event on input channel <x> will cause output trigger <n> to be asserted.

The reset behavior of this field is:

- On a External debug reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When x >= UInt(CTIDEVID.NUMCHAN), access to this field is RAZ/WI.
- Otherwise, access to this field is RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing CTIOUTEN<n>

CTIOUTEN<n> can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0x0A0 + (4 * n)`	CTIOUTEN<n>

Accessible as follows:

- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.4.30 CTIPIDR0, CTI Peripheral Identification Register 0

The CTIPIDR0 characteristics are:

Purpose

Provides information to identify a CTI component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

CTIPIDR0 is in the Debug power domain

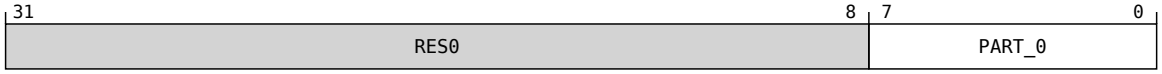
This register is required for CoreSight compliance.

Implementation of this register is OPTIONAL.

Attributes

CTIPIDR0 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

PART_0, bits [7:0]

Part number, least significant byte.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing CTIPIDR0

CTIPIDR0 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0xFE0	CTIPIDR0

Accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.4.31 CTIPIDR1, CTI Peripheral Identification Register 1

The CTIPIDR1 characteristics are:

Purpose

Provides information to identify a CTI component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

CTIPIDR1 is in the Debug power domain

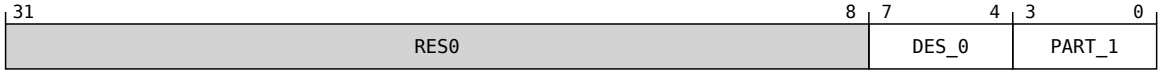
This register is required for CoreSight compliance.

Implementation of this register is OPTIONAL.

Attributes

CTIPIDR1 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

DES_0, bits [7:4]

Designer, least significant nibble of JEP106 ID code. For Arm Limited, this field is `0b1011`.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

PART_1, bits [3:0]

Part number, most significant nibble.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing CTIPIDR1

CTIPIDR1 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFE4`	CTIPIDR1

Accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.4.32 CTIPIDR2, CTI Peripheral Identification Register 2

The CTIPIDR2 characteristics are:

Purpose

Provides information to identify a CTI component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

CTIPIDR2 is in the Debug power domain

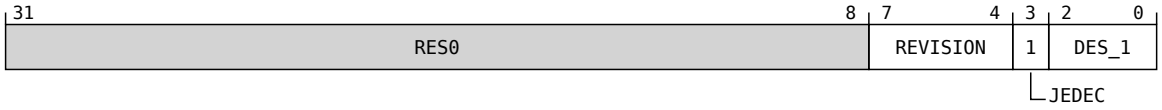
This register is required for CoreSight compliance.

Implementation of this register is OPTIONAL.

Attributes

CTIPIDR2 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

REVISION, bits [7:4]

Part major revision. Parts can also use this field to extend Part number to 16-bits.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

JEDEC, bit [3]

Indicates a JEP106 identity code is used.

Reads as `0b1`

Access to this field is RO.

DES_1, bits [2:0]

Designer, most significant bits of JEP106 ID code. For Arm Limited, this field is `0b011`.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing CTIPIDR2

CTIPIDR2 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFE8`	CTIPIDR2

Accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.4.33 CTIPIDR3, CTI Peripheral Identification Register 3

The CTIPIDR3 characteristics are:

Purpose

Provides information to identify a CTI component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

CTIPIDR3 is in the Debug power domain

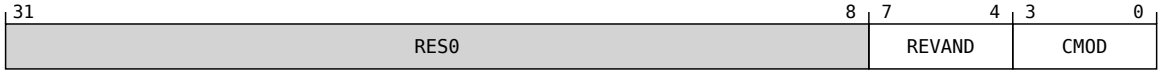
This register is required for CoreSight compliance.

Implementation of this register is OPTIONAL.

Attributes

CTIPIDR3 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

REVAND, bits [7:4]

Part minor revision. Parts using CTIPIDR2.REVISION as an extension to the Part number must use this field as a major revision number.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

CMOD, bits [3:0]

Customer modified. Indicates someone other than the Designer has modified the component.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing CTIPIDR3

CTIPIDR3 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0xFEC	CTIPIDR3

Accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.4.34 CTIPIDR4, CTI Peripheral Identification Register 4

The CTIPIDR4 characteristics are:

Purpose

Provides information to identify a CTI component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

CTIPIDR4 is in the Debug power domain

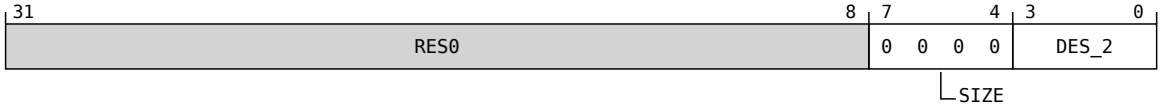
This register is required for CoreSight compliance.

Implementation of this register is OPTIONAL.

Attributes

CTIPIDR4 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

SIZE, bits [7:4]

Size of the component. Log_2 of the number of 4KB pages from the start of the component to the end of the component ID registers.

Reads as `0b0000`

Access to this field is RO.

DES_2, bits [3:0]

Designer, JEP106 continuation code, least significant nibble. For Arm Limited, this field is `0b0100`.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing CTIPIDR4

CTIPIDR4 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFD0`	CTIPIDR4

Accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.4.35 CTITRIGINSTATUS, CTI Trigger In Status register

The CTITRIGINSTATUS characteristics are:

Purpose

Provides the status of the trigger inputs.

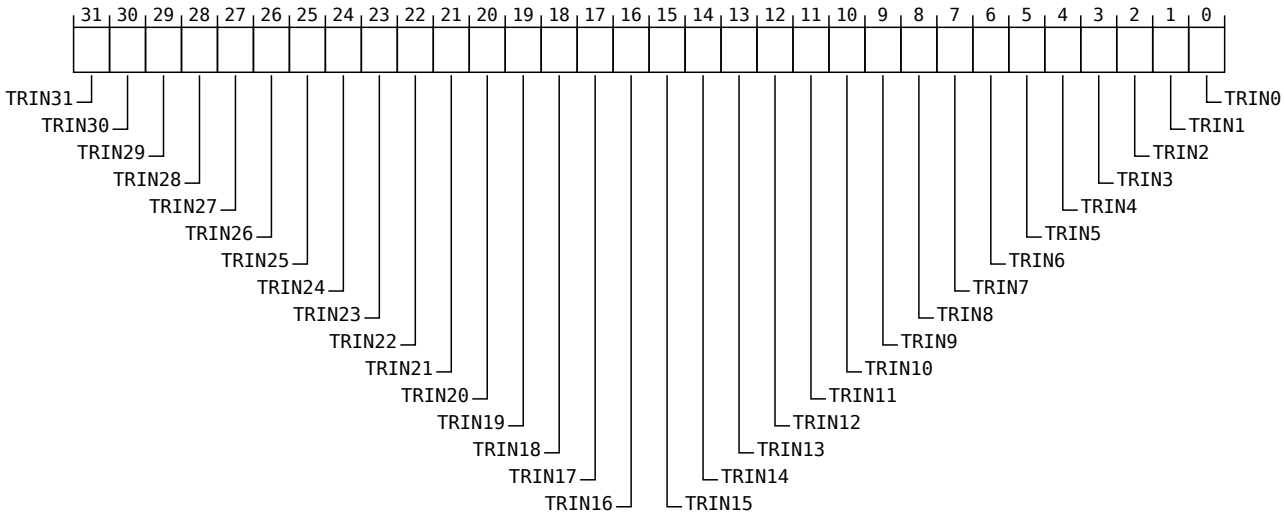
Configuration

CTITRIGINSTATUS is in the Debug power domain

Attributes

CTITRIGINSTATUS is a 32-bit register.

Field descriptions



TRIN<n> , bits [n], for n = 31 to 0

Trigger input <n> status.

TRIN<n>	Meaning
0b0	Input trigger n is inactive.
0b1	Input trigger n is active.

Not implemented and not-connected input triggers are always inactive.

It is IMPLEMENTATION DEFINED whether an input trigger that does not support multicycle events can be observed as active.

Accessing this field has the following behavior:

- When n >= UInt(CTIDEVID.NUMTRIG), access to this field is RAZ/WI.
- Otherwise, access to this field is RO.

Accessing CTITRIGINSTATUS

CTITRIGINSTATUS can be accessed through the external debug interface:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Offset	Instance
CTI	0x130	CTITRIGINSTATUS

Accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E9.4.36 CTITRIGOUTSTATUS, CTI Trigger Out Status register

The CTITRIGOUTSTATUS characteristics are:

Purpose

Provides the raw status of the trigger outputs, after processing by any IMPLEMENTATION DEFINED trigger interface logic. For output triggers that are self-acknowledging, this is only meaningful if the CTI implements multicycle channel events.

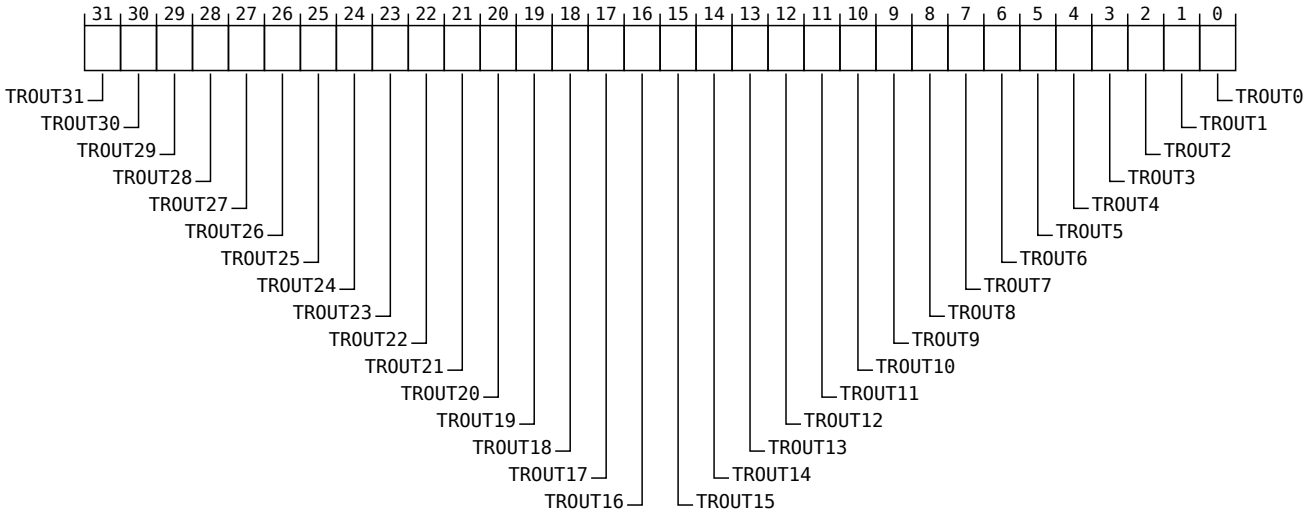
Configuration

CTITRIGOUTSTATUS is in the Debug power domain

Attributes

CTITRIGOUTSTATUS is a 32-bit register.

Field descriptions



TROUT<n> , bits [n], for n = 31 to 0

Trigger output <n> status.

If n is less than CTIDEVID.NUMTRIG, and output trigger <n> is implemented and connected, and either the trigger is not self-acknowledging or the CTI implements multicycle channel events, then permitted values for TROUT<n> are:

TROUT<n>	Meaning
0b0	Output trigger n is inactive.
0b1	Output trigger n is active.

Otherwise when n is less than CTIDEVID.NUMTRIG, it is IMPLEMENTATION DEFINED whether TROUT<n> behaves as described here or is RAZ.

Accessing this field has the following behavior:

- When n >= UInt(CTIDEVID.NUMTRIG), access to this field is RAZ/WI.
- Access to this field is RAZ/WI if all of the following are true:
  - Any of the following are true:
    - Output trigger <n> is not implemented or not connected

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— All of the following are true:
- Output trigger <n> is self-acknowledging
- CTI does not implement multicycle channel events

- an implementation implements TROUT<n>

- Otherwise, access to this field is RO.

Accessing CTITRIGOUTSTATUS

CTITRIGOUTSTATUS can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0x134	CTITRIGOUTSTATUS

Accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Part F
Memory-mapped Components of the Arm Architecture

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter F1
Requirements for Memory-mapped Components

This chapter provides some additional information about memory-mapped components. It contains the following sections:

- Supported access sizes.
- Synchronization of memory-mapped registers.
- Access requirements for reserved and unallocated registers.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F1.1 Supported access sizes

The information in this section applies to all accesses to memory-mapped components of the Armv8 and later architectures, unless a register or component description explicitly states otherwise.

The memory access sizes that are supported by any peripheral are IMPLEMENTATION DEFINED by the peripheral.

When FEAT_Debugv8p4 is implemented, each debug component has a Secure and Non-secure view. The Secure view of a debug component is mapped into Secure physical memory and the Non-secure view of a debug component is mapped into Non-secure memory. Apart from access conditions, the Non-secure and Secure views of the debug components are identical.

An implementation of a memory-mapped component that is compatible with the Armv8 and later architectures must support the following:

- Word-aligned 32-bit accesses to access 32-bit registers.
- If the system includes any component with direct memory access to the memory-mapped component which needs to access the component but does not support making 64-bit accesses, word-aligned 32-bit accesses to either half of a 64-bit register that is mapped to a doubleword-aligned pair of adjacent 32-bit locations. This includes, but is not limited to:
  - A PE that implements a 32-bit ISA that does not include 64-bit memory operations. For example, Armv8-M T32.
  - A Debug Access Port that cannot make 64-bit accesses.

  Note

  Some memory-mapped components of the Arm architecture require support for word-aligned 32-bit accesses to either half of a 64-bit memory mapped register even if all components with direct memory access to the memory-mapped component support making 64-bit accesses. These include:

  - The memory-mapped interface to the external debug and CTI registers that are described in External Debug Register Descriptions.
  - The memory-mapped interfaces to the Generic Timer registers that are described in System Level Implementation of the Generic Timer.
  - When FEAT_PMUv3_EXT32 is implemented, the memory-mapped interfaces to the Performance Monitors registers that are described in Recommended External Interface to the Performance Monitors.
- Doubleword-aligned 64-bit accesses to access 64-bit registers that are mapped to a doubleword-aligned pair of adjacent 32-bit locations.

Unless otherwise specified, all registers are only single-copy atomic at word granularity. This means that for 64-bit accesses to a 64-bit register, the system might generate a pair of 32-bit accesses. The order in which the two halves are accessed is not specified.

The following accesses are not supported:

- Byte accesses.
- Halfword accesses.
- Unaligned word accesses. These accesses are not word single-copy atomic.
- Unaligned doubleword accesses. These accesses are not doubleword single-copy atomic.
- Doubleword accesses to a pair of 32-bit locations that are not a doubleword-aligned pair that forms a 64-bit register.
- Quadword accesses or higher accesses.
- Exclusive accesses.

For unsupported accesses, it is IMPLEMENTATION DEFINED and might be UNPREDICTABLE whether:

- The access generates an External abort or not.
- The defined side-effects of a read occur or not. A read returns an IMPLEMENTATION DEFINED value that might be UNKNOWN.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- A write is ignored or sets the accessed register or registers to UNKNOWN values.
- The access generates a fault handling interrupt or not.

Such accesses are often due to software faults. That is, software has accessed a location in a way that the component does not support. See also *Arm® Reliability, Availability, and Serviceability (RAS) System Architecture, for A-profile architecture* (ARM IHI 0100) for recommended behavior on accesses that constitute software faults.

For memory-mapped accesses from a PE that complies with an Arm architecture, the single-copy atomicity rules for the instruction, the type of instruction, and the type of memory that is accessed, determine the size of the access that is made by an instruction. Example F1-1 shows this.

Example F1-1 Access sizes for memory-mapped accesses

Two Load Doubleword instructions that are made to consecutive doubleword-aligned locations generate a pair of single-copy atomic doubleword reads. However, if the accesses are made to Normal memory or Device-GRE memory they might appear as a single quadword access that is not supported by the peripheral.

The Arm architecture does not require the size of each element that is accessed by a multi-register load or store instruction to be identifiable by the memory system beyond the PE. Unless otherwise specified by the component, any access to a memory-mapped component of the Arm architecture is defined to be beyond the PE.

Software must use a Device-nGRE or stronger memory type, and only single register load and store instructions, to create memory accesses that are supported by the peripheral. For more information, see Memory types and attributes.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F1.2 Synchronization of memory-mapped registers

This section describes the synchronization requirements for the memory-mapped accesses to System registers.

This section refers to accesses to external system control registers as external reads and external writes. It refers to accesses to System registers as direct reads, direct writes, indirect reads, and indirect writes.

Note

Synchronization requirements for AArch64 System registers define direct read, direct write, indirect read, and indirect write, and classifies external reads as indirect reads and external writes as indirect writes.

Writes to the same register are serialized, meaning they are observed in the same order by all observers, although some observers might not observe all of the writes. Unless otherwise stated, external writes to different registers are not necessarily observed in the same order by all observers as the order in which they complete.

Explicit synchronization is not required for an external read or an external write by an external agent to be observable to a following external read or external write by that agent to the same register using the same address, and so is never required for registers that are accessible as external system control registers.

Unless required to be observable to all observers in finite time, without explicit synchronization, explicit synchronization is normally required following an external write to any register for that write to be observable by:

- A direct access.
- An indirect read by an instruction.
- An external read of the register using a different address.

This means that an external write by an external agent is guaranteed to have an effect on subsequent instructions executed by the PE only if all of the following are true:

- The write has completed.
- The PE has executed a Context synchronization event.
- The Context synchronization event was executed after the write completed.

The order and synchronization of direct reads and direct writes of System registers is defined by Synchronization requirements for AArch64 System registers.

The external agent must be able to guarantee completion of a write. For example, the agent can:

- Mark the memory as Device-nGnRnE and executing a DSB barrier, if the system supports this property.
- If the register is read/write and reads are not destructive, read back the value written.
- Use some guaranteed property of the connection between the PE and the external agent.

The external agent and PE can guarantee ordering by, for example, passing messages in an ordered way with respect to the external write and the Context synchronization event, and relying on the memory ordering rules provided by the memory model.

External reads and external write complete in the order in which they arrive at the PE. For accesses to different register locations, the external agent must create this order. The agent can:

- Mark the memory as Device-nGnRnE or Device-nGnRE.
- Use the appropriate memory barriers.
- Rely on some guaranteed property of the connection between the PE and the external agent.

However, the external agent cannot force the synchronization of completed writes.

In a simple sequential execution, an indirect write that occurs as a side-effect of an access happens atomically with the access, meaning no other accesses are allowed between the register access and its side-effect.

Without explicit synchronization to guarantee the order of the accesses, where the same register is accessed by two or more of a System register access instruction, and external agent, and autonomous asynchronous event, or as a result of a

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

memory-mapped access, the behavior must be as if the accesses occurred atomically and in any order. This applies even if the accesses occur simultaneously.

For example, some registers have the property that for certain bits a write of 0 is ignored and a write of 1 has an effect. This means the simultaneous writes must be merged.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F1.3 Access requirements for reserved and unallocated registers

This section describes the access requirements for reserved and unallocated memory-mapped components.

The following information relates to certain types of reserved accesses:

- Reads and writes of unallocated locations. These accesses are reserved for the architecture.
- Reads and writes of locations for features that are not implemented, including:
  - OPTIONAL features that are not implemented.
  - Breakpoints and watchpoints that are not implemented.
  - Performance Monitors counters that are not implemented.
  - CTI triggers that are not implemented.
  - Error records that are not implemented.

  These accesses are reserved.
- Reads of WO locations. These accesses are reserved for the architecture.
- Writes to RO locations. These accesses are reserved for the architecture.

Reserved accesses are normally RAZ/WI. However, software must not rely on this property as the behavior of reserved values might change in a future revision of the architecture. Software must treat reserved accesses as RES0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter F2
System Level Implementation of the Generic Timer

This chapter defines the system level implementation of the Generic Timer. It contains the following sections:

- About the Generic Timer specification.
- Memory-mapped counter module.
- Memory-mapped timer components.

Note

- Generic Timer memory-mapped register descriptions describes the System level Generic Timer registers. These registers are memory-mapped.
- Additional Information for Implementations of the Generic Timer gives additional information, that does not form part of the architectural definition of a system level implementation of the Generic Timer.
- The Generic Timer in AArch64 state gives a general description of the AArch64 state view of the Generic Timer, and describes the AArch64 System register interface to the Generic Timer.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F2.1 About the Generic Timer specification

The Generic Timer in AArch64 state describes the Arm Generic Timer and its implementation as seen from AArch64 state. These chapters include the definition of the low-latency System register interface to the Generic Timer. However, the Arm Generic Timer architecture also defines a memory-mapped component, that comprises:

- A memory-mapped counter module, that controls the generation of the Count value used by the Generic Timer. This memory-mapped counter module is required in any Arm Generic Timer implementation that requires software control of the Count value of the Generic Timer.
- Optional memory-mapped timer modules. These give a standardized way of providing timers for programmable system components other than PEs that implement the Arm architecture.

The full set of Generic Timer components summarizes these components as seen from AArch64 state. The system level components of the Generic Timer summarizes the system level components.

F2.1.1 Registers in the system level implementation of the Generic Timer

Registers that control components of the system level implementation of the Generic Timer are grouped into *frames*. This specification defines the registers in each frame, and their offsets within the frame. The system defines the position of each frame in the memory map. This means the base addresses for each frame are IMPLEMENTATION DEFINED.

Note

The final 12 words of the first or only 4KB block of a register memory frame is an ID block.

Each frame must be in its own memory page, or memory protection region, and must be aligned to the size of the translation granule or protection granule.

Note

When a system level implementation of the Generic Timer is accessed by a PE:

- Using a VMSA, each frame is in its own memory page, aligned to the size of the translation granule.
- Using a PMSA, each frame is in its own memory protection region, aligned to the size of the memory protection granule.

The following sections give more information about the requirements for the system level Generic Timer component:

- Endianness and supported access sizes.
- Power and reset domains for the system level implementation of the Generic Timer.

F2.1.1.1 Endianness and supported access sizes

All memory-mapped peripherals defined in the Arm architecture must be little-endian. This means the system-level Generic Timer registers, and the register frames, are little-endian.

The memory access sizes supported by any peripheral is IMPLEMENTATION DEFINED by the peripheral. For accesses to the memory-mapped Generic Timer registers implementations must:

- Comply with the requirements of Supported access sizes.
- Support word-aligned 32-bit accesses to access 32-bit registers or either half of a 64-bit register mapped to a doubleword-aligned pair of adjacent 32-bit locations, even if all components with direct memory access to the Generic Timer support making 64-bit accesses.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F2.1.1.2 Power and reset domains for the system level implementation of the Generic Timer

The power and reset domains of the system level implementation of the Generic Timer are IMPLEMENTATION DEFINED as part of the system implementation. In register descriptions, they are called Timer resets to indicate they can be outside the PE power and reset domains defined by the remainder of this manual.

The Arm architecture requires that the CNTCR.{FCREQ, EN} and CNTSR.FCACK fields reset to 0. These Timer reset values apply only on powerup of the power domain in which the registers are implemented or a reset of the reset domain in which they are implemented.

Every other register, or register field, of a system level implementation of the Generic Timer resets to a value that is architecturally UNKNOWN if it has a meaningful reset value. These Timer resets apply on powerup of the power domain in which the register is implemented, and on a reset of the reset domain in which it is implemented.

F2.1.2 The system level components of the Generic Timer

Each system level component has one or two register frames. The possible system level components are:

The memory-mapped counter module, required

This module controls the system counter. It has two frames:

- A control frame, CNTControlBase.
- A status frame, CNTReadBase.

Memory-mapped counter module describes this component.

The memory-mapped timer control module, required

The system level implementation of the Generic Timer can provide up to eight timers, and the memory-mapped timer control module identifies:

- Which timers are implemented.
- The features of each implemented timer.

This module has a single frame, CNTCTLBase.

The CNTCTLBase frame describes this frame.

Memory-mapped timers, optional

An implemented memory-mapped timer:

- Must provide a privileged view of the timer, in the CNTBase*N* frame.
- Optionally provides an unprivileged view of the timer in the CNTEL0Base*N* frame.

N is the timer number, and the corresponding frame number, in the range 0-7.

The CNTBaseN and CNTEL0BaseN frames describes these frames.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F2.2 Memory-mapped counter module

The memory-mapped counter module provides top-level control of the system counter. The CNTControlBase frame holds the registers for the memory-mapped counter, and provides:

- An RW control register, CNTCR, that provides:
  - An enable bit for the system counter.
  - An enable bit for Halt-on-debug. For more information, see Halt-on-debug.
  - A field that can be written to request a change to the counter resolution. This mechanism means that, for example, if the counter resolution is halved then the increment to the counter at each tick of the clock is doubled.
    For more information, see Control of counter resolution and increment.

  Writes to this register are rare.
- A RO status register, CNTSR, that provides:
  - A bit that indicates whether the system counter is halted because of an asserted Halt-on-debug signal.
  - A field that indicates whether a requested change to the counter resolution has completed.
- Two contiguous 32-bit RW registers that hold the current system counter value, CNTCV. If the system supports 64-bit atomic accesses, these two registers must be accessible by such accesses.

  The system counter must be disabled before writing to these registers, otherwise the effect of the write is UNPREDICTABLE.

  Writes to these registers are rare.
- A *Frequency modes table* of one or more 32-bit entries, where:
  - The first entry in the table defines the *base frequency* of the system counter. It is IMPLEMENTATION DEFINED whether the base frequency indicates the effective frequency of the counter, or the maximum counter resolution.
    Arm recommends that the base frequency indicates the maximum counter resolution.
  - Each subsequent entry in the table defines an alternative frequency of the system counter, that must be an exact divisor of the base frequency.

  A 32-bit zero entry immediately follows the last table entry.

  This table can be RO or RW. For more information, see The Frequency modes table.

In an implementation with multiple physical address spaces, CNTControlBase is only accessible by accesses from the most secure physical address space. For example, in a system with Secure and Non-secure memory maps, CNTControlBase is only accessible by Secure accesses.

In addition, the CNTReadBase frame includes a read-only copy of the system counter value, CNTCV, as two contiguous 32-bit RO registers. If the system supports 64-bit atomic accesses, these two registers must be accessible by such accesses.

Counter module control and status register summary describes CNTReadBase and CNTControlBase memory maps, and the registers in each frame.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F2.2.1 Control of counter resolution and increment

The system counter has a fixed *base frequency*, and must maintain the required counter accuracy, meaning Arm recommends that it does not gain or lose more than ten seconds in a 24-hour period, see The system counter. However, the counter can operate at a lower resolution than the base frequency, using a correspondingly larger increment. For example, it can increment by a value four times larger when running at a quarter of the base frequency. Any lower resolution operation, and any switching between resolutions, must not reduce the accuracy of the counter.

Control of the system counter resolution and increment is provided only through the memory-mapped counter module. The following sections describe this control:

- The Frequency modes table.
- Changing the system counter and increment.

F2.2.1.1 The Frequency modes table

The Frequency modes table starts at offset `0x20` in the CNTControlBase frame.

Table entries are 32-bits, and each entry specifies a selected frequency, in Hz.

The first entry in the table specifies the base frequency of the system counter.

When the system counter is operating at a lower frequency than the base frequency, the increment applied at each counter update is increased by the ratio of the base frequency and the selected frequency.

Note

Where the effective frequency and the base frequency are not the same, the increment is not solely defined by the ratio of the base frequency and the selected frequency. For example, a counter with an effective frequency of 1GHz and a base frequency of 125MHz will increment by a value of 8 on each tick of the base frequency. If the Frequency modes table provides an additional selected frequency of 31.25MHz, then when this frequency is selected the counter increments by 32 on each tick of the selected frequency.

A 32-bit word of zero value marks the end of the table. That is, the word of memory immediately after the last entry in the table must be zero.

The only required entry in the table is the entry for the base frequency.

Typically, the Frequency modes table is in RO memory. However, a system implementation might use RW memory for the table, and initialize the table entries as part of its startup sequence. Therefore, the CNTControlBase memory map shows the table region as RO or RW.

Arm strongly recommends that the Frequency modes table is not updated once the system is running.

The architecture can support up to 1004 entries in the Frequency modes table, including the zero-word end marker, and the number of entries is IMPLEMENTATION DEFINED, up to this limit.

Note

- Arm considers it likely that implementations will require significantly fewer entries than the architectural limit.
- In the CNTControlBase frame, the offset range `0x0C0-0x0FC` can be used for IMPLEMENTATION DEFINED registers. If any registers are defined in this space, then the Frequency modes table cannot extend beyond offset `0x0B8`, with a zero word at offset `0x0BC`. This means that if any IMPLEMENTATION DEFINED registers are defined the maximum number of entries in the table is 40, including the zero-word end marker.

F2.2.1.2 Changing the system counter and increment

The value of the CNTCR.FCREQ field specifies which entry in the Frequency modes table specifies the selected frequency.

Changing the value of CNTCR.FCREQ requests a change to the selected frequency. To ensure the frequency change does not affect the overall accuracy of the counter, a change is made as follows:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When changing from a higher frequency to a lower frequency, the counter:
  1. Continues running at the higher frequency until the count reaches an integer multiple of the required lower frequency.
  2. Switches to operating at the lower frequency.
- When changing from a lower frequency to a higher frequency, the counter:
  1. Waits until the end of the current lower-frequency cycle.
  2. Makes the counter increment required for operation at that lower frequency.
  3. Switches to operating at the higher frequency.

When the frequency has changed, CNTSR is updated to indicate the new selected frequency. Therefore, a system component that is waiting for a frequency change can poll CNTSR to detect the change.

F2.2.2 Halt-on-debug

The CNTCR register provides an enable bit for an OPTIONAL Halt-on-debug signal.

When the CNTCR.HDBG bit is set to 1, and the Halt-on-debug signal is implemented and asserted, the system counter is halted. Otherwise, the system counter ignores the state of this signal.

Where the system counter implements a Halt-on-debug signal and the system supports halting the system counter, Arm recommends that the Halt-on-debug signal can be controlled by a debugger using the Embedded Cross-Trigger (ECT) using a system-level cross-trigger interface that includes:

- A debug request output trigger event that asserts the Halt-on-debug signal.
- A restart request output trigger event that deasserts the Halt-on-debug signal.

For more information, see About the Embedded Cross-Trigger.

Note

Software must use the Halt-on-debug enable bit to ensure that the timers cannot be halted maliciously in an attempt to prohibit progress.

For more information about Halt-on-debug, contact Arm.

F2.2.3 Counter module control and status register summary

The Counter module control and status registers are memory-mapped registers in the following register memory *frames*:

- A control frame, with base address CNTControlBase.
- A status frame, with base address CNTReadBase.

Each of these register memory frames is in its own memory page or memory protection region, and the frame base address points to the start of this region. Each base address must be aligned to the size of the translation granule or protection granule.

Note

Each frame of a memory-mapped Generic Timer takes the name of its base address.

In each register memory frame, the memory at offset `0xFD0-0xFFF` is reserved for twelve 32-bit IMPLEMENTATION DEFINED ID registers, see the CounterID<n> register descriptions for more information.

Note

The Arm architecture requires memory-mapped peripherals to be little-endian, and therefore the counter is little-endian.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

In an implementation with multiple physical address spaces, CNTControlBase is only accessible by accesses from the most secure physical address space. For example, in a system with Secure and Non-secure memory maps, CNTControlBase is only accessible by Secure accesses.

Table F2-1 shows the CNTControlBase control registers, in order of their offsets from the CNTControlBase base address, for an implementation that includes registers in the IMPLEMENTATION DEFINED register space 0x0C0-0x0FC, and also has fewer than 39 CNTFID<n> registers. The Frequency modes table describes how this memory map differs if more CNTFID<n> registers are implemented.

Generic Timer memory-mapped register descriptions describes each of these registers.

Table F2-1 CNTControlBase memory map

Offset	Name	Type	Description
0x000	CNTCR	RW	Counter Control Register.
0x004	CNTSR	RO	Counter Status Register.
0x008	CNTCV[31:0]	RW	Counter Count Value register.
0x00C	CNTCV[63:32]	RW	
0x010	CNTSCR[a]	RW	Counter Scale Register.
0x014-0x018	-	RES0	Reserved.
0x01C	CNTID[a]	RO	Counter Identification Register.
0x020	CNTFID0	RO or RW	Frequency modes table, and end marker. For more information, see The Frequency modes table.
0x020+4*n*	CNTFID<n>	RO or RW	
0x024+4*n*	-	RO or RW, RAZ	
(0x028+4*n*)-0x0BC	-	RO, RES0	Reserved.
0x0C0-0x0FC	-	IMPLEMENTATION DEFINED	Reserved for IMPLEMENTATION DEFINED registers.
0x100-0xFCC	-	RO, RES0	Reserved.
0xFD0-0xFFC	CounterID<n>	RO	Counter ID registers 0-11.

a. Implemented only if FEAT_CNTSC is implemented.

Table F2-2 shows the CNTReadBase control registers, in order of their offsets from the CNTReadBase base address. Generic Timer memory-mapped register descriptions describes each of these registers.

Table F2-2 CNTReadBase memory map

Offset	Name	Type	Description
0x000	CNTCV[31:0]	RO	Counter Count Value register
0x004	CNTCV[63:32]	RO	
0x008-0xFCC	-	RES0	Reserved
0xFD0-0xFFC	CounterID<n>	RO	Counter ID registers 0-11

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F2.3 Memory-mapped timer components

This part of the Arm Generic Timer specification defines an optional memory-mapped timer component. This can be implemented as part of any programmable system component that does not incorporate a System register mapped Arm Generic Timer, to provide that system component with the timer functionality of an Arm Generic Timer.

The memory map consists of up to eight timer *frames*. The base address of a frame is CNTBaseN, where *N* numbers from 0 up to a maximum permitted value of 7.

Each CNTBaseN timer frame:

- Provides its own set of timers and associated interrupts.
- Is implemented in its own memory page or memory protection region.
- Is implemented at a base address, identified as CNTBaseN, that is aligned to the size of the translation granule or memory protection region.

For each implemented CNTBaseN frame the system can optionally provide an unprivileged view of the frame, described as the EL0 view of the frame. The base address of this second view of the CNTBaseN frame is CNTEL0BaseN.

Note

In the naming of the registers associated with a CNTBaseN or CNTEL0BaseN frame, the value of *N* is represented as <n>, for example CNTACR<n>.

If a CNTEL0BaseN frame is implemented:

- Is implemented in its own memory page or memory protection region and is aligned to the size of the translation granule or memory protection region.
- All registers visible in CNTBaseN, except for CNTVOFF and CNTEL0ACR, can be visible in CNTEL0BaseN.
  - Control fields in CNTEL0ACR determine whether each register is visible.
- The offsets of all visible registers are the same as their offsets in the CNTBaseN frame.

In addition to the implemented CNTBaseN and CNTEL0BaseN frames, the system must provide a single control frame at base address CNTCTLBase. CNTCTLBase must be implemented in its own memory page or memory protection region and is aligned to the size of the translation granule or memory protection region.

The system defines the position of each frame in the memory map. This means the values of each of the CNTBaseN, CNTEL0BaseN, and CNTCTLBase base addresses is IMPLEMENTATION DEFINED.

Note

The Arm architecture requires memory-mapped peripherals to be little-endian, and therefore the memory-mapped timers are little-endian.

The following sections describe the implementation of a memory-mapped view of the counter and timer:

- The CNTCTLBase frame.
- The CNTBaseN and CNTEL0BaseN frames.

Note

Providing a complete set of features in a system level implementation gives an implementation example for a system level Generic Timer implementation that provides equivalent features to a System registers Generic Timer implementation in a PE that includes all of the Exception levels.

F2.3.1 The CNTCTLBase frame

The CNTCTLBase frame contains:

- An identification register for the features of the memory-mapped counter and timer implementation.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Access controls for each CNTBaseN frame.
- A virtual offset register for frames that implement a virtual timer.

Table F2-3 shows the CNTCTLBase registers, in order of their offsets from the CNTCTLBase base address.

Note

CNTFRQ and CNTVOFF registers are also implemented in a System register interface to the Generic Timer.

Generic Timer memory-mapped register descriptions describes each of these registers.

Table F2-3 CNTCTLBase memory map

Offset	Register	Type	Security[a]	Description
0x000	CNTFRQ[b]	RW	Restricted	Counter Frequency register.
0x004	CNTNSAR	RW	Restricted	Counter Non-Secure Access register.
0x008	CNTTIDR	RO	No restrictions	Counter Timer ID register.
0x00C- 0x03F	-	RES0	-	Reserved.
0x040+4*N*[c]	CNTACR<n>	RW	Programmable[d]	Counter Access Control register *N*.
0x060- 0x07F	-	RES0	-	Reserved.
0x080+8*N*[c]	CNTVOFF<n>[31:0][b]	RW[e]	Programmable[d]	Virtual Offset register *N*. If the CNTBaseN frame has virtual timer capability then CNTVOFF<n> is implemented as an RW register, otherwise its location is RAZ/WI.
0x084+8*N*[c]	CNTVOFF<n>[63:32][b]	RW[e]		
0x0C0-0x0FC	-	RES0	-	Reserved.
0x100-0x7FC	-	-	-	IMPLEMENTATION DEFINED.
0x800-0xFBC	-	RES0	-	Reserved.
0xFC0-0xFCF	-	-	-	IMPLEMENTATION DEFINED.
0xFD0- 0xFFC	CounterID<n>	RO	No restrictions	Counter ID registers 0-11.

a. Access security restrictions in an implementation with multiple Security states. See the individual register descriptions and CNTNSAR for details on the restrictions. In an implementation that supports only a single Security state all registers are accessible as shown in the *Type* column.

b. These registers are also defined in the System register interface to the Generic Timer, and therefore are also described in Generic Timer registers and Generic Timer registers. The bit assignments of the registers are identical in the System register interface and in the memory-mapped system level interface.

c. Implemented for each value of *N* from 0 to 7 for which a CNTBaseN frame is implemented.

d. The CNTNSAR determines the Non-secure accessibility of the CNTACR<n> and the CNTVOFF<n> in the CNTCTLBase frame. For more information, see the register descriptions.

e. Address is reserved, RAZ/WI if register not implemented.

All implementations of the Generic Timer include the virtual counter. Therefore, conceptually, all implementations include the CNTVOFF register that defines the virtual offset between the physical count and the virtual count. If a memory-mapped Generic Timer component does not distinguish between real time and virtual time, then it can implement CNTVOFF as RAZ/WI. Otherwise CNTVOFF is an RW register, and Arm strongly recommends that the system only permits access to CNTVOFF from EL2 or higher.

F2.3.2 The CNTBaseN and CNTEL0BaseN frames

Each CNTBaseN frame, or {CNTBaseN, CNTEL0BaseN} pair of frames, provides a memory-mapped counter and timer, see:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The CNTBaseN frame.
- The CNTEL0BaseN frame.
- CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames.

F2.3.2.1 The CNTBase*N* frame

Table F2-4 shows the CNTBaseN registers, in order of their offsets from the CNTBaseN base address. Whether a frame includes a virtual timer is IMPLEMENTATION DEFINED. If it does not, then memory at offsets `0x030-0x03C` is RAZ/WI. Except for CNTEL0ACR and the CounterID<n> registers, equivalent registers are also implemented in a System register interface to the timer component of a Generic Timer.

Generic Timer memory-mapped register descriptions describes each of these registers.

Table F2-4 CNTBase*N* memory map

Offset	Register	Type	Description
`0x000`	CNTPCT[31:0][a]	RO	Physical Count register.
`0x004`	CNTPCT[63:32][a]	RO	
`0x008`	CNTVCT[31:0][a]	RO	Virtual Count register.
`0x00C`	CNTVCT[63:32][a]	RO	
`0x010`	CNTFRQ[a]	RO[c]	Counter Frequency register.
`0x014`	CNTEL0ACR	RW[b]	Counter EL0 Access Control Register, optional in the CNTBaseN memory map.
`0x018`	CNTVOFF[31:0][a]	RO[c]	Virtual Offset register. If CNTVOFF in the CNTCTLBase frame is an RW register, a read of this register returns the value of that register. Otherwise is RAZ.
`0x01C`	CNTVOFF[63:32][a]	RO[c]	
`0x020`	CNTP_CVAL[31:0][a]	RW	Physical Timer CompareValue register.
`0x024`	CNTP_CVAL[63:32][a]	RW	
`0x028`	CNTP_TVAL[a]	RW	Physical TimerValue register.
`0x02C`	CNTP_CTL[a]	RW	Physical Timer Control register.
`0x030`	CNTV_CVAL[31:0][a]	RW[b]	Virtual Timer CompareValue register, optional in the CNTBaseN memory map.
`0x034`	CNTV_CVAL[63:32][a]	RW[b]	
`0x038`	CNTV_TVAL[a]	RW[b]	Virtual TimerValue register, optional in the CNTBaseN memory map.
`0x03C`	CNTV_CTL[a]	RW[b]	Virtual Timer Control register, optional in the CNTBaseN memory map.
`0x040-0xFCF`	-	RES0	Reserved.
`0xFD0-0xFFC`	CounterID<n>	RO	Counter ID registers 0-11.

a. These registers are also defined in the System register interface to the Generic Timer, and therefore are also described in Generic Timer registers and Generic Timer registers. The bit assignments of the registers are identical in the System register interface and in the memory-mapped system level interface.

b. Address is reserved, RAZ/WI if register not implemented.

c. The CNTCTLBase frame includes an RW view of this register.

F2.3.2.2 The CNTEL0Base*N* frame

For any value of *N*, the layout of the registers in the CNTEL0Base*N* frame is identical to the CNTBaseN frame, except that, in the CNTEL0Base*N* frame:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- CNTVOFF is never visible, and the memory at `0x018-0x01C` is RAZ/WI.
- CNTEL0ACR is never visible, and the memory at `0x014` is RAZ/WI.
- If implemented in the CNTBaseN frame, CNTEL0ACR controls whether CNTPCT, CNTVCT, CNTFRQ, the Physical Timer, and the Virtual Timer registers are visible in the CNTEL0BaseN frame.
  If CNTEL0ACR is not implemented then these registers are not visible in the CNTEL0BaseN frame, and their addresses in that frame are RAZ/WI.

If an implementation supports 64-bit atomic accesses, then CNTPCT, CNTVCT, CNTVOFF, CNTP_CVAL, and CNTV_CVAL must be accessible as atomic 64-bit values.

F2.3.2.3 CNTCTLBase status and control fields for the CNTBase*N* and CNTEL0Base*N* frames

In the CNTCTLBase frame:

CNTTIDR controls:

- Whether each CNTBaseN frame is implemented.
- If a CNTBaseN frame is implemented, whether:
  - That CNTBaseN frame has virtual timer capability.
  - A corresponding CNTEL0BaseN frame is implemented.

CNTNSAR controls:

Whether all the following are accessible by Non-secure accesses:

- CNTACR<n>.
- CNTVOFF<n>.
- Each implemented CNTBaseN frame, and any corresponding CNTEL0BaseN frame.

The CNTACR<n> registers control:

For each implemented CNTBaseN frame, the accessibility of the following registers in that frame:

- CNTP_CTL, CNTP_CVAL, and CNTP_TVAL.
- CNTV_CTL, CNTV_CVAL, and CNTV_TVAL.
- CNTVOFF.
- CNTFRQ.
- CNTPCT.
- CNTVCT.

For CNTACR<n>, the value of <n> corresponds to the value of *N* for the controlled CNTBaseN frame.

The CNTVOFF<n> registers provide:

For each implemented CNTBaseN frame that has virtual capability, the RW copy of the CNTVOFF register for that frame.

Note

In a CNTBaseN frame that has virtual timer capability the CNTVOFF register is RO.

For CNTVOFF<n>, the value of <n> corresponds to the value of *N* for the controlled CNTBaseN frame.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter F3
Recommended External Interface to the Performance Monitors

This chapter describes the recommended external interface to the Performance Monitors. It contains the following section:

- About the external interface to the Performance Monitors registers.

Note

Performance Monitors external register descriptions describes the external view of the Performance Monitors registers.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F3.1 About the external interface to the Performance Monitors registers

Arm recommends that:

- An implementation provides the OPTIONAL external debug interface to the Performance Monitors registers, FEAT_PMUv3_EXT.

  Note

  A debugger can use this interface to access counters in the Performance Monitors.

- The implementation includes the OPTIONAL support for memory-mapped access to the External debug interface.

  Note

  — Software running on any PE in a system can use this interface to access counters in the Performance Monitors.

  — Privileged software should use the MMU to control access to this interface.

- The external debug interface is implemented as defined in Recommended External Debug Interface.

The following sections describe the memory-mapped views of the Performance Monitors registers:

- Differences in the external views of the Performance Monitors registers.
- Synchronization of changes to the memory-mapped views.
- Access permissions for external views of the Performance Monitors.

In this section, unless the context explicitly indicates otherwise, any reference to *a memory-mapped view* applies equally to a register view using:

- An access through an external debug interface.
- A memory-mapped access.

F3.1.1 Endianness and supported access sizes

When an implementation supports memory-mapped access to the external debug interface the interface is accessed as a little-endian memory-mapped peripheral. External Performance Monitors registers summary gives the memory map of these registers.

The memory access sizes supported by any peripheral is IMPLEMENTATION DEFINED by the peripheral. For accesses to the external interface to the Performance Monitors registers implementations must:

- Comply with the requirements of Supported access sizes.
- When FEAT_PMUv3_EXT32 is implemented, support word-aligned 32-bit accesses to access 32-bit registers or either half of a 64-bit register that is mapped to a doubleword-aligned pair of adjacent 32-bit locations, even if all components with direct memory access to the PMU support making 64-bit accesses.

Permitted word-aligned 32-bit accesses are single-copy atomic at word granularity. When FEAT_PMUv3_EXT64 is implemented, permitted doubleword-aligned 64-bit accesses are single-copy atomic at doubleword granularity.

F3.1.2 Differences in the external views of the Performance Monitors registers

An external view of the Performance Monitors registers accesses the same registers as the System registers interface described in Performance Monitors Extension registers, except that:

- The PMSELR is accessible only in the System registers interface.
- The following registers are accessible only in external views:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - PMPCSR.
  - PMCFGR.
  - PMIIDR.
  - PMLAR.
  - PMLSR.
  - PMAUTHSTATUS.
  - PMDEVARCH.
  - PMDEVTYPE.
  - PMPIDR0.
  - PMPIDR1.
  - PMPIDR2.
  - PMPIDR3.
  - PMPIDR4.
  - PMCIDR0.
  - PMCIDR1.
  - PMCIDR2.
  - PMCIDR3.
  - PMEVFILT2R<n>.

  Performance Monitors external register descriptions describes these registers.

- The following registers are accessible only in the external view when FEAT_PMUv3_EXT32 is implemented:
  - PMCID1SR.
  - PMVIDSR.
  - PMCID2SR.
  - PMDEVAFF0.
  - PMDEVAFF1.

  Performance Monitors external register descriptions describes these registers.
- The following controls do not affect the external views:
  - PMSELR.
  - PMUSERENR.
  - HDCR.{TPM, TPMCR, HPMN}.

  Instead, see the register descriptions in External System Control Register Descriptions.
- The PMSWINC_EL0 register is OPTIONAL in the external view when FEAT_PMUv3_EXT32 is implemented.

F3.1.3 Synchronization of changes to the memory-mapped views

Synchronization must comply with Synchronization of memory-mapped registers.

In particular, if a Performance Monitor is visible in both System register and an external view, and is accessed simultaneously through these two mechanisms, the behavior must be as if the accesses occurred atomically and in any order. For more information, see Synchronization of changes to the external debug registers.

F3.1.4 Access permissions for external views of the Performance Monitors

For more information, see External debug interface register access permissions.

The following table shows the access permissions for the Performance Monitors registers in a Debug implementation for Armv8-R implementations.

The table uses the following terms:

DLK When FEAT_DoubleLock is implemented and locked, `DoubleLockStatus()` == TRUE, accesses to some registers produce an error. Applies to both interfaces.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EPMAD When `AllowExternalPMUAccess()` == FALSE, external debug access is disabled for the access. See also Behavior of a not permitted memory-mapped access.

EPMSSAD If FEAT_PMUv3_SS is implemented, when `AllowExternalPMSSAccess()` == FALSE, external debug access is disabled for the access. See also Behavior of a not permitted memory-mapped access.

Error Indicates that the access gives an error response.

Def This shows the default access permissions, if none of the conditions in this list prevent access to the register.

Off The Core power domain is completely off, or in a low-power state where the Core power domain registers cannot be accessed, and EDPRSR.PU will read as zero.

Note

If debug power is off, then all external debug interface accesses return an error.

OSLK When the OS Lock is locked, OSLAR_EL1.OSLK == 1, accesses to some registers produces an error. This column shows the effect of this control on accesses using the external debug interface.

SLK This indicates the modified default access permissions for OPTIONAL memory-mapped accesses to the external debug interface if the optional Software Lock is locked. See Register access permissions for memory-mapped accesses.

For all other accesses, this column is ignored.

Note

When FEAT_PMUv3_EXT64 is implemented, the Software Lock is not implemented.

- Indicates that the control has no effect on the behavior of the access:

- If no other control affects the behavior, the Default access behavior applies.
- However, another control might determine the behavior.

Table F3-1 Access permissions for the Performance Monitors registers when FEAT_PMUv3_EXT32 is implemented

Offset	Register	Domain	Off	DLK	OSLK	EPMAD	EPMSSAD	Def	SLK
`0x000+8xn` `0x004+8xn`	PMEVCNTR<n>_EL0[31:0][a] PMEVCNTR<n>_EL0[63:32][a]	Core	Error	Error	Error	Error	-	RW	RO
`0x0F8` `0x0FC`	PMCCNTR_EL0[31:0] PMCCNTR_EL0[63:32]	Core	Error	Error	Error	Error	-	RW	RO
`0x200` `0x204`	PMPCSR[31:0] PMPCSR[63:32]	Core	Error	Error	Error	-	-	RO	RO
`0x208`	PMCID1SR	Core	Error	Error	Error	-	-	RO	RO
`0x20C`	PMVIDSR	Core	Error	Error	Error	-	-	RO	RO
`0x220` `0x224`	PMPCSR[31:0] PMPCSR[63:32]	Core	Error	Error	Error	-	-	RO	RO
`0x228`	PMCID1SR	Core	Error	Error	Error	-	-	RO	RO
`0x22C`	PMCID2SR	Core	Error	Error	Error	-	-	RO	RO
`0x400+4xn`	PMEVTYPER<n>_EL0[a][31:0]	Core	Error	Error	Error	Error	-	RW	RO
`0x47C`	PMCCFILTR_EL0[31:0]	Core	Error	Error	Error	Error	-	RW	RO
`0xC00` `0xC04`	PMCNTENSET_EL0[31:0] PMCNTENSET_EL0[63:32]	Core	Error	Error	Error	Error	-	RW	RO

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Offset	Register	Domain	Off	DLK	OSLK	EPMAD	EPMSSAD	Def	SLK
0xC20 0xC24	PMCNTENCLR_EL0[31:0] PMCNTENCLR_EL0[63:32]	Core	Error	Error	Error	Error	-	RW	RO
0xC40 0xC44	PMINTENSET_EL1[31:0] PMINTENSET_EL1[63:32]	Core	Error	Error	Error	Error	-	RW	RO
0xC60 0xC64	PMINTENCLR_EL1[31:0] PMINTENCLR_EL1[63:32]	Core	Error	Error	Error	Error	-	RW	RO
0xC80 0xC84	PMOVSCLR_EL0[31:0] PMOVSCLR_EL0[63:32]	Core	Error	Error	Error	Error	-	RW	RO
0xCA0	PMSWINC_EL0	Core	Error	Error	Error	Error	-	WO	WI
0xCC0 0xCC4	PMOVSSET_EL0[31:0] PMOVSSET_EL0[63:32]	Core	Error	Error	Error	Error	-	RW	RO
0xD80- 0xDFC	-	-	IMPLEMENTATION DEFINED registers[d]						
0xE00	PMCFGR	Core	Error	Error	Error	Error	-	RO	RO
0xE04	PMCR_EL0	Core	Error	Error	Error	Error	-	RW	RO
0xE08	PMIIDR	Core	Error	Error	Error	Error	-	RO	RO
0xE20 0xE24 0xE28 0xE2C	PMCEID0 PMCEID1 PMCEID2 PMCEID3	Core	Error	Error	Error	Error	-	RO	RO
0xE38- 0xE3C		-	IMPLEMENTATION DEFINED registers[d]						
0xE40 0xE44	PMMIR[31:0] PMMIR[63:32]	Core	Error	Error	Error	Error	-	RO	RO
0xE80- 0xEFC	Integration registers	-	IMPLEMENTATION DEFINED registers[d]						
0xF00- 0xFFC	Management registers and CoreSight compliance								

a. Implemented event counters only. *n* is the counter number.

b. See IMPLEMENTATION DEFINED registers.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter F4
External System Control Register Descriptions

This chapter describes the external system control registers. It excludes the External debug registers that are described in External Debug Register Descriptions. It contains the following sections:

- About the external system control register descriptions.
- External Performance Monitors registers summary.
- Performance Monitors external register descriptions.
- Generic Timer memory-mapped registers overview.
- Generic Timer memory-mapped register descriptions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.1 About the external system control register descriptions

This chapter describes the external system control registers other than the external debug registers. That is, it describes:

An external view of the Performance Monitors registers

Arm recommends that implementations provide access to the Performance Monitors registers through the OPTIONAL External debug interface, and provide the OPTIONAL memory-mapped interface to this interface:

- External Performance Monitors registers summary lists the registers that are accessible in this view of the Performance Monitors, and describes their memory map.
- Performance Monitors external register descriptions describes each of the memory-mapped registers.

Recommended External Interface to the Performance Monitors describes the recommended interface to these registers.

Note

The Performance Monitors Extension describes the Performance Monitors. The following section describes the System register interfaces to the Performance Monitors, Performance Monitors registers, for accesses from an Exception level that is using AArch64.

The registers for the system level Generic Timer component

Any implementation that includes the Generic Timer must include the memory-mapped system level component described in System Level Implementation of the Generic Timer. In this chapter:

- Generic Timer memory-mapped registers overview gives an overview of the registers, referring to System Level Implementation of the Generic Timer for more information.
- Generic Timer memory-mapped register descriptions describes each of the memory-mapped registers.

Note

The Generic Timer in AArch64 state describes the Generic Timer component that is accessible using the System registers. The following section describes the System register interfaces to that component Generic Timer registers, for accesses from an Exception level that is using AArch64.

Note

External Debug Register Descriptions describes the external debug registers.
Arm® Memory System Resource Partitioning and Monitoring (MPAM) System Component Specification* (ARM IHI 0099)* describes the MPAM memory-mapped registers.
Arm® Reliability, Availability, and Serviceability (RAS) System Architecture, for A-profile architecture (ARM IHI 0100) describes the RAS memory-mapped registers.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.2 External Performance Monitors registers summary

When an implementation provides access to the Performance Monitors registers through the External debug interface, that interface provides access to:

- Performance Monitors System registers.
- A read-only configuration register, PMCFGR.
- The OPTIONAL CoreSight registers for the Performance Monitors, if they are implemented.

The locations of the registers are defined as offsets from a system-defined base address. Performance Monitors external register views defines this memory map.

F4.2.1 Performance Monitors external register views

The following tables show the external view of the Performance Monitors registers:

- Table F4-1 when FEAT_PMUv3_EXT32 is implemented.

All other entries are reserved.

Note

- Counters that are reserved because HDCR.HPMN has been changed from its reset value remain visible in any external view.
- The registers that relate to an implemented event counter, PMN*x*, are PMEVCNTR<n> and PMEVTYPER<n>.
- Tables in this section only list the Armv8 registers. For encoding information of the registers introduced by Armv9, see the individual register descriptions in this chapter.

Each entry in the Name column links to the register description in Performance Monitors external register descriptions.

Table F4-1 Performance Monitors external register views when FEAT_PMUv3_EXT32 is implemented

Name	Type	Description	Offset
PMEVCNTR<n>_EL0[31:0] PMEVCNTR<n>_EL0[63:32]	RW	Performance Monitors Event Counter Register	0x000+8xn 0x004+8xn
PMCCNTR_EL0[31:0] PMCCNTR_EL0[63:32]	RW	Performance Monitors Cycle Counter Register[a]	0x0F8 0x0FC
PMPCSR[31:0][b] PMPCSR[63:32][b]	RW	Program Counter Sample Register	0x200 0x204
PMCID1SR[b]	RW	CONTEXTIDR_EL1 Sample Register	0x208
PMVIDSR[b]	RW	VMID Sample Register	0x20C
PMPCSR[31:0][b] PMPCSR[63:32][b]	RW	Program Counter Sample Register, alias	0x220 0x224
PMCID1SR[b]	RW	CONTEXTIDR_EL1 Sample Register (alias)	0x228
PMCID2SR[b]	RW	CONTEXTIDR_EL2 Sample Register	0x22C
PMEVTYPER<n>_EL0 [31:0]	RW	Performance Monitors Event Type and Filter Register	0x400+4xn
PMCCFILTR_EL0 [31:0]	RW	Performance Monitors Cycle Counter Filter Register	0x47C
-	-	Reserved	0x480- 0x7FC[c]
PMEVFILT2R<n>	RW	Performance Monitors Event Filter Registers	0x800+8xn

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Name	Type	Description	Offset
PMEVTYPER<n>_EL0 [63:32]	RW	Performance Monitors Event Type and Filter Register	0xA00+4xn[d]
PMCCFILTR_EL0 [63:32]	RW	Performance Monitors Cycle Counter Filter Register	0xA7C[d]
-	-	Reserved	0xA80-0xAFC[d]
-	-	IMPLEMENTATION DEFINED	0xB00-0xBFC
PMCNTENSET_EL0[31:0][e] PMCNTENSET_EL0[63:32] [h]	RW	Performance Monitors Count Enable Set Register	0xC00 0xC04
PMCNTENCLR_EL0[31:0][e] PMCNTENCLR_EL0[63:32] [e]	RW	Performance Monitors Count Enable Clear Register	0xC20 0xC24
PMINTENSET_EL1[31:0][e] PMINTENSET_EL1[63:32] [e]	RW	Performance Monitors Interrupt Enable Set Register	0xC40 0xC44
PMINTENCLR_EL1[31:0][e] PMINTENCLR_EL1[63:32] [e]	RW	Performance Monitors Interrupt Enable Clear Register	0xC60 0xC64
PMOVSCLR_EL0[31:0][e] PMOVSCLR_EL0[63:32][e]	RW	Performance Monitors Overflow Flag Status Clear Register	0xC80 0xC84
PMSWINC_EL0[f]	WO	Performance Monitors Software Increment Register	0xCA0
-	-	Reserved	0xCA0-0xCA4[d]
PMOVSSET_EL0[31:0][e] PMOVSSET_EL0[63:32][e]	RW	Performance Monitors Overflow Flag Status Set Register	0xCC0 0xCC4
-	-	Reserved	0xCE0
-	-	IMPLEMENTATION DEFINED	0xD80-0xDFC
PMCFGR	RO	Performance Monitors Configuration Register	0xE00
PMCR_EL0	RW	Performance Monitors Control Register	0xE04
PMIIDR	RO	Performance Monitors Implementation Identification Register	0xE08
PMCEID0 PMCEID1 PMCEID2 PMCEID3	RO	Performance Monitors Common Event Identification Registers	0xE20 0xE24 0xE28 0xE2C
-	-	Reserved	0xE30-0xE34[c]
-	-	IMPLEMENTATION DEFINED	0xE38-0xE3C
PMMIR[31:0][e] PMMIR[63:32][e]	RO	Performance Monitors Machine Identification Register, bits [31:0]	0xE40 0xE44
-	-	Reserved	0xE50-0xE54
-	-	IMPLEMENTATION DEFINED	0xE80-0xEFC
PMITCTRL[g]	RW	Integration Model Control registers	0xF00
PMDEVAFF0[g] PMDEVAFF1[g]	RO	Device Affinity registers	0xFA8 0xFAC
PMLAR[g] , [h]	WO	Lock Access Register	0xFB0
PMLSR[g] , [h]	RO	Lock Status Register	0xFB4
PMAUTHSTATUS[g]	RO	Authentication Status Register	0XFB8
PMDEVARCH[g]	RO	Device Architecture Register	0xFBC
PMDEVID[b]	RO	Performance Monitors Device ID Register	0xFC8
PMDEVTYPE[g]	RO	Device Type Register	0xFCC

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Name	Type	Description	Offset
PMPIDR4[g]	RO	Peripheral ID registers	0xFD0
PMPIDR0[g]			0xFE0
PMPIDR1[g]			0xFE4
PMPIDR2[g]			0xFE8
PMPIDR3[g]			0xFEC
PMCIDR0[g]	RO	Component ID registers	0xFF0
PMCIDR1[g]			0xFF4
PMCIDR2[g]			0xFF8
PMCIDR3[g]			0xFFC

a. The interface must support at least single-copy atomic 32-bit accesses. If single-copy atomic 64-bit access to the registers is not possible, software must use a high-low-high read access to read the counter value if the counter is enabled.

b. PC Sample-based Profiling Extension registers. Implemented only when FEAT_PCSRv8p2 is implemented, except that from Armv8.2 PMDEVID is required regardless of whether FEAT_PCSRv8p2 is implemented.

Before Armv8.2. the PC Sample-based Profiling Extension can, instead, be implemented in the memory-mapped debug registers space, see The PC Sample-based Profiling Extension.

c. OPTIONAL.

d. CoreSight interface registers, see Management registers and CoreSight compliance.

e. The Software lock registers are defined as part of CoreSight compliance, but their contents depend on the type of access that is made and whether the OPTIONAL Software lock is implemented. See the register description for details.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.3 Performance Monitors external register descriptions

This section describes the external view of the Performance Monitors registers. External Performance Monitors registers summary lists these registers in offset order.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.3.1 PMAUTHSTATUS, Performance Monitors Authentication Status register

The PMAUTHSTATUS characteristics are:

Purpose

Provides information about the state of the IMPLEMENTATION DEFINED authentication interface for Performance Monitors.

Configuration

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

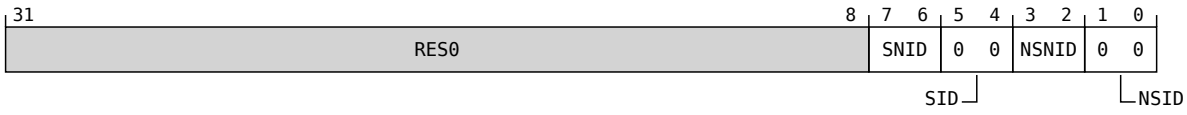
This register is OPTIONAL, and is required for CoreSight compliance. Arm recommends that this register is implemented.

This register is present only when FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMAUTHSTATUS are RES0.

Attributes

PMAUTHSTATUS is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

SNID, bits [7:6]

Holds the same value as DBGAUTHSTATUS_EL1.SNID.

SID, bits [5:4]

Secure invasive debug.

SID	Meaning
0b00	Not implemented.

All other values are reserved.

Access to this field is RO.

NSNID, bits [3:2]

Holds the same value as DBGAUTHSTATUS_EL1.NSNID.

NSID, bits [1:0]

Non-secure invasive debug.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NSID	Meaning
0b00	Not implemented.

All other values are reserved.

Access to this field is RO.

Accessing PMAUTHSTATUS

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset 0xFB8 from PMU

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.3.2 PMCCFILTR_EL0, Performance Monitors Cycle Counter Filter Register

The PMCCFILTR_EL0 characteristics are:

Purpose

Determines the modes in which the Cycle Counter, PMCCNTR_EL0, increments.

Configuration

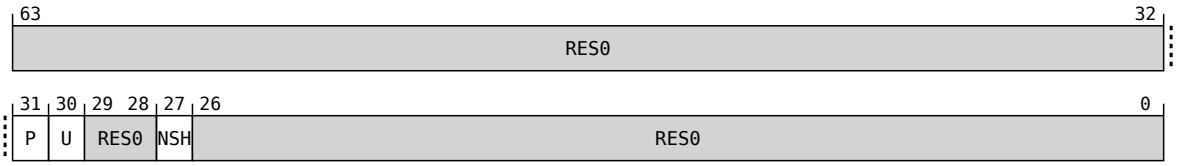
PMCCFILTR_EL0 is in the Core power domain.

This register is present only when FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMCCFILTR_EL0 are RES0.

Attributes

PMCCFILTR_EL0 is a 64-bit register.

Field descriptions



Bits [63:32]

Reserved, RES0.

P, bit [31]

EL1 filtering. Controls counting cycles in EL1.

P	Meaning
0b0	This mechanism has no effect on filtering of cycles.
0b1	The PE does not count cycles in EL1.

If Secure and Non-secure states are implemented, then counting cycles in Non-secure EL1 is further controlled by PMCCFILTR_EL0.NSK.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

U, bit [30]

EL0 filtering. Controls counting cycles in EL0.

U	Meaning
0b0	This mechanism has no effect on filtering of cycles.
0b1	The PE does not count cycles in EL0.

If Secure and Non-secure states are implemented, then counting cycles in Non-secure EL0 is further controlled by PMCCFILTR_EL0.NSU.

The reset behavior of this field is:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [29:28]

Reserved, RES0.

NSH, bit [27]

EL2 filtering. Controls counting cycles in EL2.

NSH	Meaning
0b0	The PE does not count cycles in EL2.
0b1	This mechanism has no effect on filtering of cycles.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [26:0]

Reserved, RES0.

Accessing PMCCFILTR_EL0

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented

[31:0] Accessible at offset 0x47C from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.3.3 PMCCNTR_EL0, Performance Monitors Cycle Counter

The PMCCNTR_EL0 characteristics are:

Purpose

Holds the value of the processor Cycle Counter, CCNT, that counts processor clock cycles. For more information, see 'Time as measured by the Performance Monitors cycle counter'.

PMCCFILTR_EL0 determines the modes and states in which the PMCCNTR_EL0 can increment.

Configuration

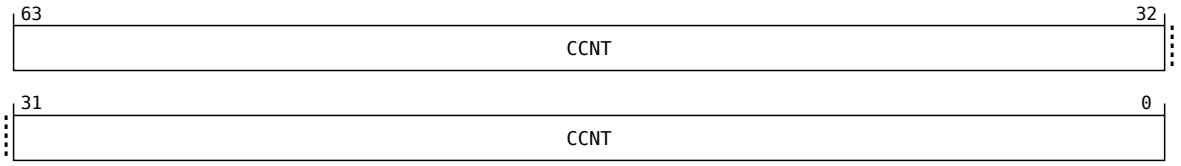
PMCCNTR_EL0 is in the Core power domain.

This register is present only when FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMCCNTR_EL0 are RES0.

Attributes

PMCCNTR_EL0 is a 64-bit register.

Field descriptions



CCNT, bits [63:0]

Cycle count. Depending on the values of PMCR_EL0.{LC,D}, the cycle count increments in one of the following ways:

- Every processor clock cycle.
- Every 64th processor clock cycle.

Writing 1 to PMCR_EL0.C sets this field to 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PMCCNTR_EL0

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented

[31:0] Accessible at offset 0x0F8 from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_PMUv3_EXT32 is implemented

[63:32] Accessible at offset `0x0FC` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.3.4 PMCEID0, Performance Monitors Common Event Identification register 0

The PMCEID0 characteristics are:

Purpose

Defines which Common architectural events and Common microarchitectural events are implemented, or counted, using PMU events in the range `0x0000` to `0x001F`.

For more information about the Common events and the use of the PMCEIDn registers, see 'The PMU event number space and common events'.

Use of this register is deprecated.

Note

This view of the register was previously called PMCEID0_EL0.

Configuration

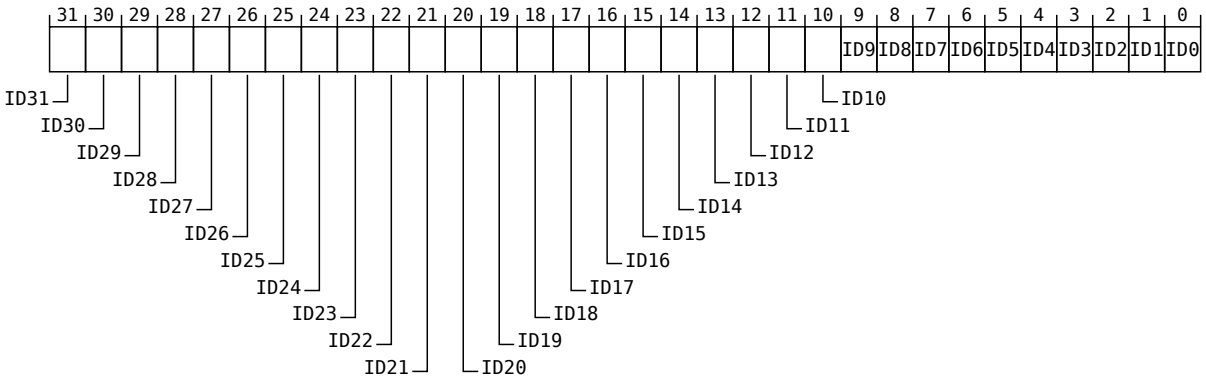
PMCEID0 is in the Core power domain.

This register is present only when FEAT_PMUv3_EXT32 is implemented. Otherwise, direct accesses to PMCEID0 are RES0.

Attributes

PMCEID0 is a 32-bit register.

Field descriptions



ID<n> , bits [n], for n = 31 to 0

ID[n] corresponds to Common event n.

For each bit:

ID<n>	Meaning
`0b0`	The Common event is not implemented, or not counted.
`0b1`	The Common event is implemented.

When the value of a bit in the field is 1, the corresponding Common event is implemented and counted.

Note

Arm recommends that if a Common event is never counted, the value of the corresponding bit is 0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional Common event.

Note

Such an event might be added retrospectively to an earlier version of the PMU architecture, provided the event does not require any additional PMU features and has an event number that can be represented in the PMCEID<n> registers of that earlier version of the PMU architecture.

Accessing PMCEID0

Note

AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented

Accessible at offset `0xE20` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.3.5 PMCEID1, Performance Monitors Common Event Identification register 1

The PMCEID1 characteristics are:

Purpose

Defines which Common architectural events and Common microarchitectural events are implemented, or counted, using PMU events in the range `0x020` to `0x03F`.

For more information about the Common events and the use of the PMCEIDn registers, see 'The PMU event number space and common events'.

Use of this register is deprecated.

Note

This view of the register was previously called PMCEID1_EL0.

Configuration

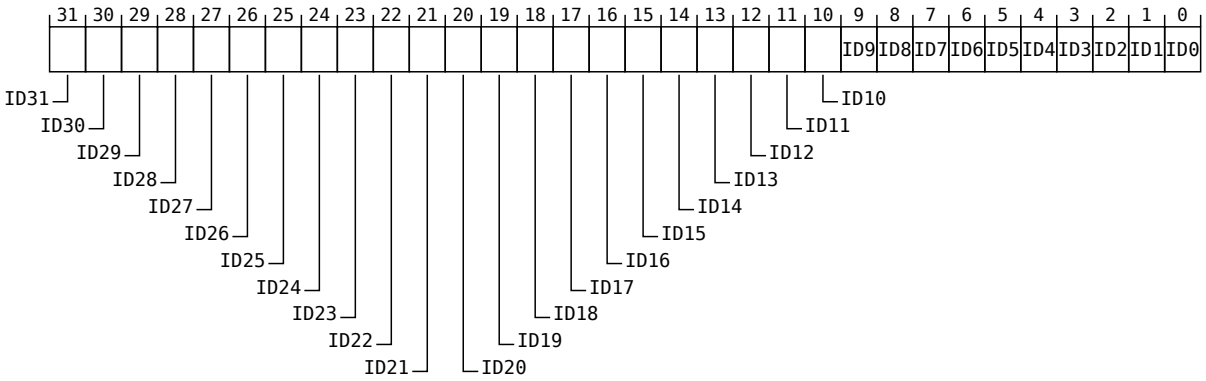
PMCEID1 is in the Core power domain.

This register is present only when FEAT_PMUv3_EXT32 is implemented. Otherwise, direct accesses to PMCEID1 are RES0.

Attributes

PMCEID1 is a 32-bit register.

Field descriptions



ID<n> , bits [n], for n = 31 to 0

ID[n] corresponds to Common event (`0x0020` + n).

For each bit:

ID<n>	Meaning
`0b0`	The Common event is not implemented, or not counted.
`0b1`	The Common event is implemented.

When the value of a bit in the field is 1, the corresponding Common event is implemented and counted.

Note

Arm recommends that if a Common event is never counted, the value of the corresponding bit is 0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional Common event.

Note

Such an event might be added retrospectively to an earlier version of the PMU architecture, provided the event does not require any additional PMU features and has an event number that can be represented in the PMCEID<n> registers of that earlier version of the PMU architecture.

Accessing PMCEID1

Note

AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented

Accessible at offset `0xE24` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.3.6 PMCEID2, Performance Monitors Common Event Identification register 2

The PMCEID2 characteristics are:

Purpose

Defines which Common architectural events and Common microarchitectural events are implemented, or counted, using PMU events in the range `0x4000` to `0x401F`.

For more information about the Common events and the use of the PMCEIDn registers, see 'The PMU event number space and common events'.

Use of this register is deprecated.

Configuration

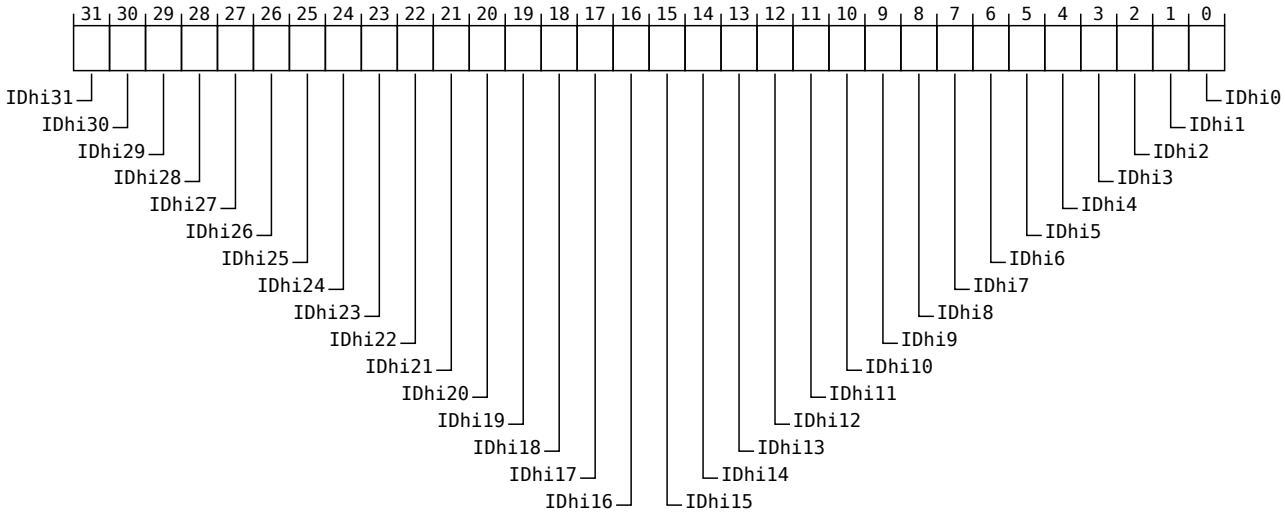
PMCEID2 is in the Core power domain.

This register is present only when FEAT_PMUv3_EXT32 is implemented and FEAT_PMUv3p1 is implemented. Otherwise, direct accesses to PMCEID2 are RES0.

Attributes

PMCEID2 is a 32-bit register.

Field descriptions



IDhi<n> , bits [n], for n = 31 to 0

IDhi[n] corresponds to Common event (`0x4000` + n).

For each bit:

IDhi<n>	Meaning
`0b0`	The Common event is not implemented, or not counted.
`0b1`	The Common event is implemented.

When the value of a bit in the field is 1, the corresponding Common event is implemented and counted.

Note

Arm recommends that if a Common event is never counted, the value of the corresponding bit is 0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional Common event.

Note

Such an event might be added retrospectively to an earlier version of the PMU architecture, provided the event does not require any additional PMU features and has an event number that can be represented in the PMCEID<n> registers of that earlier version of the PMU architecture.

Accessing PMCEID2

Note

AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented and FEAT_PMUv3p1 is implemented

Accessible at offset `0xE28` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.3.7 PMCEID3, Performance Monitors Common Event Identification register 3

The PMCEID3 characteristics are:

Purpose

Defines which Common architectural events and Common microarchitectural events are implemented, or counted, using PMU events in the range 0x4020 to 0x403F.

For more information about the Common events and the use of the PMCEIDn registers, see 'The PMU event number space and common events'.

Use of this register is deprecated.

Configuration

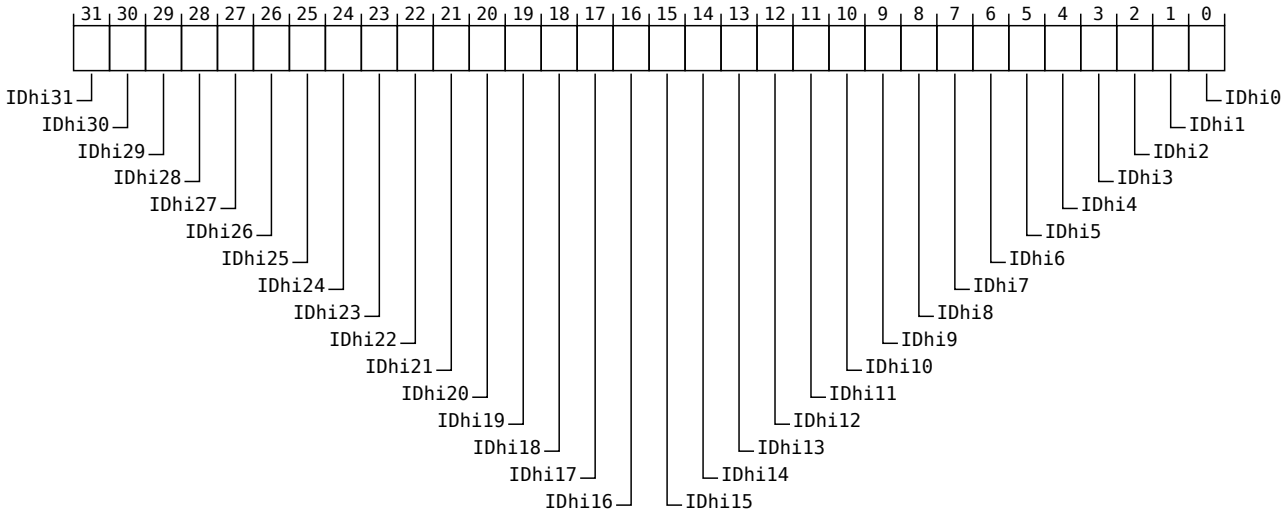
PMCEID3 is in the Core power domain.

This register is present only when FEAT_PMUv3_EXT32 is implemented and FEAT_PMUv3p1 is implemented. Otherwise, direct accesses to PMCEID3 are RES0.

Attributes

PMCEID3 is a 32-bit register.

Field descriptions



IDhi<n> , bits [n], for n = 31 to 0

IDhi[n] corresponds to Common event (0x4020 + n).

For each bit:

IDhi<n>	Meaning
0b0	The Common event is not implemented, or not counted.
0b1	The Common event is implemented.

When the value of a bit in the field is 1, the corresponding Common event is implemented and counted.

Note

Arm recommends that if a Common event is never counted, the value of the corresponding bit is 0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional Common event.

Note

Such an event might be added retrospectively to an earlier version of the PMU architecture, provided the event does not require any additional PMU features and has an event number that can be represented in the PMCEID<n> registers of that earlier version of the PMU architecture.

Accessing PMCEID3

Note

AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented and FEAT_PMUv3p1 is implemented

Accessible at offset `0xE2C` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.3.8 PMCFGR, Performance Monitors Configuration Register

The PMCFGR characteristics are:

Purpose

Contains PMU-specific configuration data.

Configuration

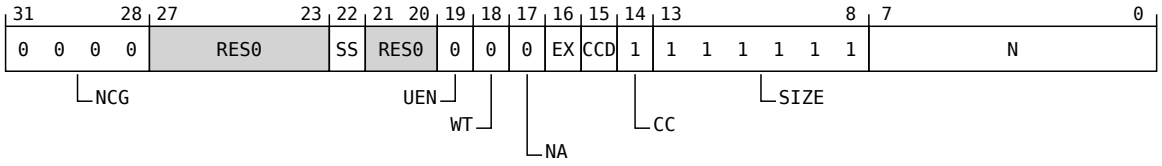
PMCFGR is in the Core power domain.

This register is present only when FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMCFGR are RES0.

Attributes

PMCFGR is a 32-bit register.

Field descriptions



NCG, bits [31:28]

Defines the number of counter groups implemented, minus one.

NCG	Meaning
0b0000	One counter group implemented.

All other values are reserved.

Access to this field is RO.

Bits [27:23]

Reserved, RES0.

SS, bit [22]

Snapshot supported.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SS	Meaning
0b0	Snapshot mechanism not supported. The locations 0x600-0x7FC and 0xE30-0xE3C are IMPLEMENTATION DEFINED.
0b1	Snapshot mechanism supported. Locations 0x600-0x7FC and 0xE30-0xE3C contain IMPLEMENTATION DEFINED snapshot registers.

If the architecture-defined form of snapshot is not implemented, a PMU might include an IMPLEMENTATION DEFINED snapshot mechanism, including one using the IMPLEMENTATION DEFINED registers 0x600-0x7FC and 0xE30-0xE3C.

Access to this field is RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [21:20]

Reserved, RES0.

UEN, bit [19]

User-mode Enable Register supported. PMUSERENR_EL0 is not visible in the external debug interface, so this bit is RAZ.

Reads as 0b0

Access to this field is RO.

WT, bit [18]

This feature is not supported, so this bit is RAZ.

Reads as 0b0

Access to this field is RO.

NA, bit [17]

This feature is not supported, so this bit is RAZ.

Reads as 0b0

Access to this field is RO.

EX, bit [16]

Export supported.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EX	Meaning
0b0	PMCR_EL0.X is RES0.
0b1	PMCR_EL0.X is read/write.

Access to this field is RO.

CCD, bit [15]

Cycle counter has prescale.

This field is RAZ

The value of this field is an IMPLEMENTATION DEFINED choice of:

CCD	Meaning
0b0	PMCR_EL0.D is RES0.
0b1	PMCR_EL0.D is read/write.

Access to this field is RO.

CC, bit [14]

Dedicated cycle counter (counter 31) supported.

Reads as 0b1

Access to this field is RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SIZE, bits [13:8]

Size of counters, minus one. This field defines the size of the largest counter implemented by the Performance Monitors Unit.

From Armv8.0, the largest counter is 64-bits, so the value of this field is `0b111111`.

This field is used by software to determine the spacing of the counters in the memory-map. From Armv8.0, the counters are a doubleword-aligned addresses.

Reads as `0b111111`

Access to this field is RO.

N, bits [7:0]

Number of counters implemented, minus one.

The value of this field is an IMPLEMENTATION DEFINED choice of:

N	Meaning
`0x00`	Only PMCCNTR_EL0 implemented.
`0x01..0x1F`	Number of counters implemented, 2 to 32, minus one.

All other values are reserved.

The count includes the cycle counter, PMCCNTR_EL0.

For example, if PMCFGR.N == `0x07`, then:

- There are eight counters in total.
- This comprises 7 event counters and the cycle counter.

The maximum number of event counters is 31.

Access to this field is RO.

Accessing PMCFGR

Note

AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented

[31:0] Accessible at offset `0xE00` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.3.9 PMCID1SR, CONTEXTIDR_EL1 Sample Register

The PMCID1SR characteristics are:

Purpose

Contains the sampled value of CONTEXTIDR_EL1, captured on reading PMPCSR[31:0].

Configuration

PMCID1SR is in the Core power domain.

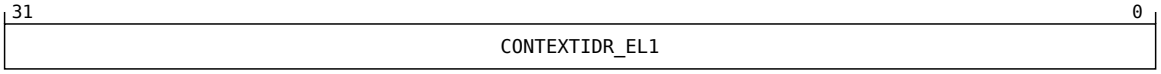
This register is a PC Sample-based Profiling Extension register.

This register is present only when FEAT_PMUv3_EXT32 is implemented and FEAT_PCSRv8p2 is implemented.

Attributes

PMCID1SR is a 32-bit register.

Field descriptions



CONTEXTIDR_EL1, bits [31:0]

Context ID. The value of CONTEXTIDR that is associated with the most recent PMPCSR sample. When the most recent PMPCSR sample is generated:

- If EL1 is using AArch64, then the Context ID is sampled from CONTEXTIDR_EL1.

Because the value written to this register is an indirect read of CONTEXTIDR, it is CONSTRAINED UNPREDICTABLE whether this register is set to the original or new value if PMPCSR samples:

- An instruction that writes to CONTEXTIDR.
- The next Context synchronization event.
- Any instruction executed between these two instructions.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing PMCID1SR

IMPLEMENTATION DEFINED extensions to external debug might make the value of this register UNKNOWN, see 'Permitted behavior that might make the PC Sample-based profiling registers UNKNOWN'.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented and FEAT_PCSRv8p2 is implemented

Accessible at offset `0x208` from PMU

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register generate an error response.
- When OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

When FEAT_PMUv3_EXT32 is implemented and FEAT_PCSRv8p2 is implemented

Accessible at offset `0x228` from PMU

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register generate an error response.
- When OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.3.10 PMCID2SR, CONTEXTIDR_EL2 Sample Register

The PMCID2SR characteristics are:

Purpose

Contains the sampled value of CONTEXTIDR_EL2, captured on reading PMPCSR[31:0].

Configuration

PMCIDR2SR is in the Core power domain.

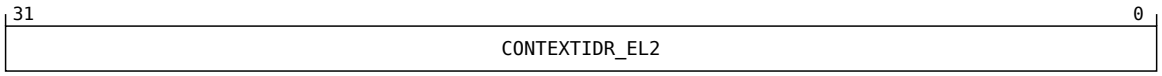
This register is a PC Sample-based Profiling Extension register.

This register is present only when FEAT_PMUv3_EXT32 is implemented and FEAT_PCSRv8p2 is implemented.

Attributes

PMCID2SR is a 32-bit register.

Field descriptions



CONTEXTIDR_EL2, bits [31:0]

Context ID. The value of CONTEXTIDR_EL2 that is associated with the most recent PMPCSR sample. When the most recent PMPCSR sample is generated:

- If the PE is not executing at EL3, EL2 is using AArch64, and EL2 is enabled in the current Security state, then this field is set to the Context ID sampled from CONTEXTIDR_EL2.
- Otherwise, this field is set to an UNKNOWN value.

Because the value written to this field is an indirect read of CONTEXTIDR_EL2, it is CONSTRAINED UNPREDICTABLE whether this field is set to the original or new value if PMPCSR samples:

- An instruction that writes to CONTEXTIDR_EL2.
- The next Context synchronization event.
- Any instruction executed between these two instructions.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing PMCID2SR

If FEAT_PMUv3_EXT64 is implemented, then the same content is present in the same location, and can be accessed using PMCCIDSR[63:32].

IMPLEMENTATION DEFINED extensions to external debug might make the value of this register UNKNOWN, see 'Permitted behavior that might make the PC Sample-based profiling registers UNKNOWN'.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented and FEAT_PCSRv8p2 is implemented

Accessible at offset 0x22C from PMU

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register generate an error response.
- When OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.3.11 PMCIDR0, Performance Monitors Component Identification Register 0

The PMCIDR0 characteristics are:

Purpose

Provides information to identify a Performance Monitor component.

For more information, see ‘About the Component Identification scheme’.

Configuration

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

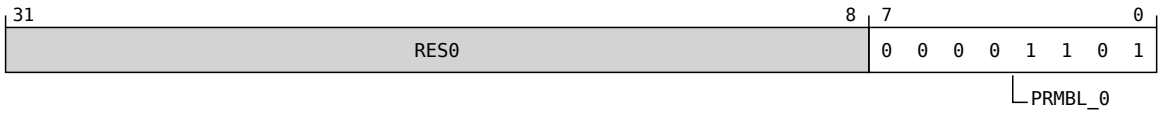
This register is required for CoreSight compliance.

This register is present only when FEAT_PMUv3_EXT is implemented and an implementation implements PMCIDR0. Otherwise, direct accesses to PMCIDR0 are RES0.

Attributes

PMCIDR0 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

PRMBL_0, bits [7:0]

Preamble.

Reads as `0x0D`

Access to this field is RO.

Accessing PMCIDR0

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset `0xFF0` from PMU

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.3.12 PMCIDR1, Performance Monitors Component Identification Register 1

The PMCIDR1 characteristics are:

Purpose

Provides information to identify a Performance Monitor component.

For more information, see ‘About the Component Identification scheme’.

Configuration

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

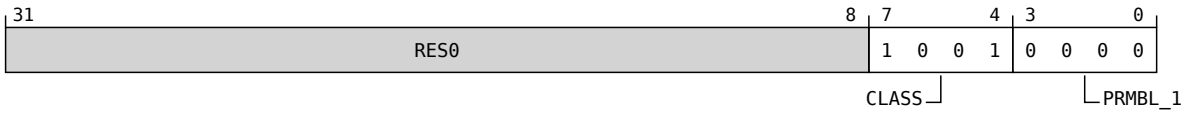
This register is required for CoreSight compliance.

This register is present only when FEAT_PMUv3_EXT is implemented and an implementation implements PMCIDR1. Otherwise, direct accesses to PMCIDR1 are RES0.

Attributes

PMCIDR1 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

CLASS, bits [7:4]

Component class.

CLASS	Meaning
0b1001	CoreSight component.

Other values are defined by the CoreSight Architecture.

This field reads as 0x9.

Access to this field is RO.

PRMBL_1, bits [3:0]

Preamble.

Reads as 0b0000

Access to this field is RO.

Accessing PMCIDR1

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset 0xFF4 from PMU

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.3.13 PMCIDR2, Performance Monitors Component Identification Register 2

The PMCIDR2 characteristics are:

Purpose

Provides information to identify a Performance Monitor component.

For more information, see ‘About the Component Identification scheme’.

Configuration

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

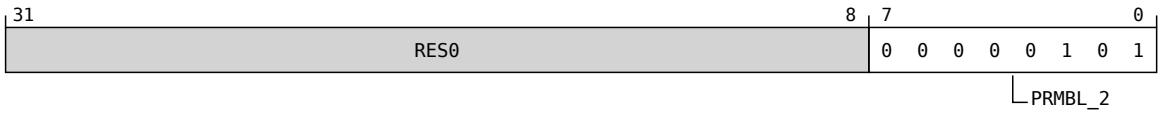
This register is required for CoreSight compliance.

This register is present only when FEAT_PMUv3_EXT is implemented and an implementation implements PMCIDR2. Otherwise, direct accesses to PMCIDR2 are RES0.

Attributes

PMCIDR2 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

PRMBL_2, bits [7:0]

Preamble.

Reads as `0x05`

Access to this field is RO.

Accessing PMCIDR2

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset `0xFF8` from PMU

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.3.14 PMCIDR3, Performance Monitors Component Identification Register 3

The PMCIDR3 characteristics are:

Purpose

Provides information to identify a Performance Monitor component.

For more information, see ‘About the Component Identification scheme’.

Configuration

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

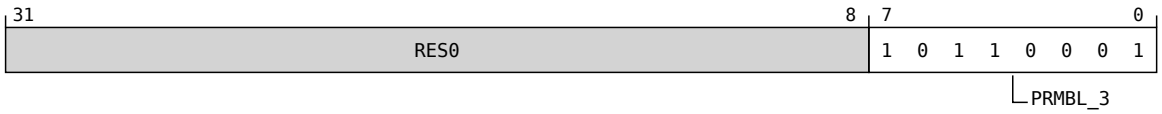
This register is required for CoreSight compliance.

This register is present only when FEAT_PMUv3_EXT is implemented and an implementation implements PMCIDR3. Otherwise, direct accesses to PMCIDR3 are RES0.

Attributes

PMCIDR3 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

PRMBL_3, bits [7:0]

Preamble.

Reads as 0xB1

Access to this field is RO.

Accessing PMCIDR3

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset 0xFFC from PMU

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.3.15 PMCNTENCLR_EL0, Performance Monitors Count Enable Clear Register

The PMCNTENCLR_EL0 characteristics are:

Purpose

Allows software to disable the following counters:

- The cycle counter PMCCNTR_EL0.
- The event counters PMEVCNTR<n>_EL0.

Reading from this register shows which counters are enabled.

Configuration

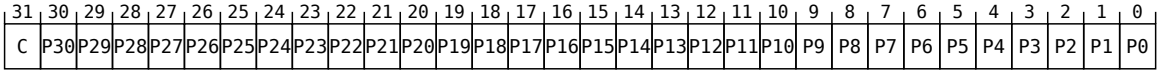
PMCNTENCLR_EL0 is in the Core power domain.

This register is present only when FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMCNTENCLR_EL0 are RES0.

Attributes

PMCNTENCLR_EL0 is a 32-bit register.

Field descriptions



C, bit [31]

PMCCNTR_EL0 disable. On writes, allows software to disable PMCCNTR_EL0. On reads, returns the PMCCNTR_EL0 enable status.

C	Meaning
0b0	PMCCNTR_EL0 disabled.
0b1	PMCCNTR_EL0 enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is W1C.

P<m> , bits [m], for m = 30 to 0

PMEVCNTR<m>_EL0 disable. On writes, allows software to disable PMEVCNTR<m>_EL0. On reads, returns the PMEVCNTR<m>_EL0 enable status.

P<m>	Meaning
0b0	PMEVCNTR<m>_EL0 disabled.
0b1	PMEVCNTR<m>_EL0 enabled.

The reset behavior of this field is:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= NUM_PMU_COUNTERS, access to this field is RAZ/WI.
- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is W1C.

Accessing PMCNTENCLR_EL0

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented

[31:0] Accessible at offset `0xC20` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.3.16 PMCNTENSET_EL0, Performance Monitors Count Enable Set Register

The PMCNTENSET_EL0 characteristics are:

Purpose

Allows software to enable the following counters:

- The cycle counter PMCCNTR_EL0.
- The event counters PMEVCNTR<n>_EL0.

Reading from this register shows which counters are enabled.

Configuration

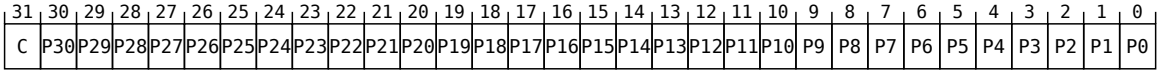
PMCNTENSET_EL0 is in the Core power domain.

This register is present only when FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMCNTENSET_EL0 are RES0.

Attributes

PMCNTENSET_EL0 is a 32-bit register.

Field descriptions



C, bit [31]

PMCCNTR_EL0 enable. On writes, allows software to enable PMCCNTR_EL0. On reads, returns the PMCCNTR_EL0 enable status.

C	Meaning
0b0	PMCCNTR_EL0 disabled.
0b1	PMCCNTR_EL0 enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is W1S.

P<m> , bits [m], for m = 30 to 0

PMEVCNTR<m>_EL0 enable. On writes, allows software to enable PMEVCNTR<m>_EL0. On reads, returns the PMEVCNTR<m>_EL0 enable status.

P<m>	Meaning
0b0	PMEVCNTR<m>_EL0 disabled.
0b1	PMEVCNTR<m>_EL0 enabled.

The reset behavior of this field is:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= NUM_PMU_COUNTERS, access to this field is RAZ/WI.
- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is W1S.

Accessing PMCNTENSET_EL0

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented

[31:0] Accessible at offset `0xC00` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.3.17 PMCR_EL0, Performance Monitors Control Register

The PMCR_EL0 characteristics are:

Purpose

Configures and controls the Performance Monitors counters.

Configuration

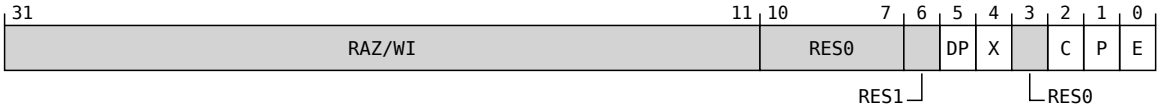
PMCR_EL0 is in the Core power domain.

This register is present only when FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMCR_EL0 are RES0.

Attributes

PMCR_EL0 is a 32-bit register.

Field descriptions



Bits [31:11]

Reserved, RAZ/WI.

Hardware must implement this field as RAZ/WI. Software must not rely on the register reading as zero, and must use a read-modify-write sequence to write to the register.

Bits [10:7]

Reserved, RES0.

Bit [6]

Reserved, RES1.

DP, bit [5]

When FEAT_PMUv3p1 is implemented:

Disable cycle counter when event counting is prohibited.

DP	Meaning
0b0	Cycle counting by PMCCNTR_EL0 is not affected by this mechanism.
0b1	Cycle counting by PMCCNTR_EL0 is disabled in prohibited regions: • If FEAT_PMUv3p1 is implemented, EL2 is implemented, and MDCR_EL2.HPMD or HDCR.HPMD is 1, then cycle counting by PMCCNTR_EL0 is disabled at EL2.

The conditions when this field disables the cycle counter are the same as when event counting by an event counter in the first range is prohibited. For more information about event counter ranges, see MDCR_EL2.HPMN.

For more information, see ‘Prohibiting event and cycle counting’.

The reset behavior of this field is:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

X, bit [4]

When the implementation includes a PMU event export bus:

Enable export of events in an IMPLEMENTATION DEFINED PMU event export bus.

X	Meaning
0b0	Do not export events.
0b1	Export events where not prohibited.

This field enables the exporting of events over an IMPLEMENTATION DEFINED PMU event export bus to another device.

No events are exported when counting is prohibited.

This field does not affect the generation of Performance Monitors overflow interrupt requests or signaling to a cross-trigger interface (CTI) that can be implemented as signals exported from the PE.

If FEAT_ETMv4 is implemented, this field does affect the use of PMU events as an External Input by the trace unit.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

Bit [3]

Reserved, RES0.

C, bit [2]

Cycle counter reset. The effects of writing to this field are:

C	Meaning
0b0	No action.
0b1	Reset PMCCNTR_EL0 to zero.

Note

Resetting PMCCNTR_EL0 does not change the cycle counter overflow field. The value of PMCR_EL0.LC is ignored, and bits [63:0] of the cycle counter are reset.

Access to this field is WO/RAZ.

P, bit [1]

Event counter reset.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

P	Meaning
0b0	No action.
0b1	Reset all event counters PMEVCNTR<n>_EL0 to zero.

Writes to this field do not affect the cycle counter PMCCNTR_EL0.

Note

Resetting the event counters does not change the event counter overflow fields.

Access to this field is WO/RAZ.

E, bit [0]

Enable.

E	Meaning
0b0	Affected counters are disabled and do not count.
0b1	Affected counters are enabled by PMCNTENSET_EL0.

The counters affected by this field are:

- The event counters in the first range. For more information about event counter ranges, see MDCR_EL2.HPMN.
- The cycle counter PMCCNTR_EL0.

Other event counters are not affected by this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Accessing PMCR_EL0

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented

Accessible at offset 0xE04 from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.3.18 PMDEVAFF0, Performance Monitors Device Affinity register 0

The PMDEVAFF0 characteristics are:

Purpose

Copy of the low half of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the Performance Monitor component relates to.

Configuration

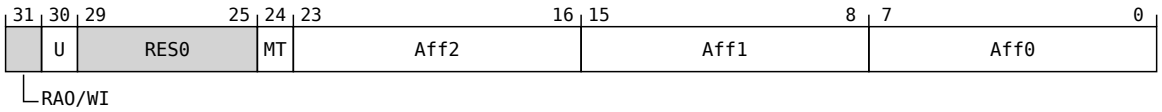
If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is present only when FEAT_PMUv3_EXT32 is implemented.

Attributes

PMDEVAFF0 is a 32-bit register.

Field descriptions



Bit [31]

Reserved, RAO/WI.

U, bit [30]

Indicates a Uniprocessor system, as distinct from PE 0 in a multiprocessor system.

The value of this field is an IMPLEMENTATION DEFINED choice of:

U	Meaning
0b0	Processor is part of a multiprocessor system.
0b1	Processor is part of a uniprocessor system.

Access to this field is RO.

Bits [29:25]

Reserved, RES0.

MT, bit [24]

Indicates whether the lowest level of affinity consists of logical PEs that are implemented using an interdependent approach, such as multithreading. See the description of Aff0 for more information about affinity levels.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MT	Meaning
0b0	Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is largely independent.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MT	Meaning
0b1	Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is very interdependent.

Note

This field does not indicate that multithreading is implemented and does not indicate that PEs with different affinity level 0 values, and the same values for affinity level 1 and higher are implemented.

Access to this field is RO.

Aff2, bits [23:16]

Affinity level 2. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Aff1, bits [15:8]

Affinity level 1. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Aff0, bits [7:0]

Affinity level 0. The value of the MPIDR.{Aff2, Aff1, Aff0} or MPIDR_EL1.{Aff3, Aff2, Aff1, Aff0} set of fields of each PE must be unique within the system as a whole.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing PMDEVAFF0

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented

Accessible at offset 0xFA8 from PMU

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.3.19 PMDEVAFF1, Performance Monitors Device Affinity register 1

The PMDEVAFF1 characteristics are:

Purpose

Copy of the high half of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the Performance Monitor component relates to.

Configuration

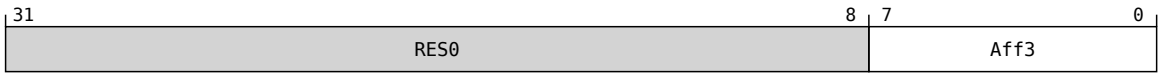
If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is present only when FEAT_PMUv3_EXT32 is implemented.

Attributes

PMDEVAFF1 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

Aff3, bits [7:0]

Affinity level 3. See the description of PMDEVAFF0.Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing PMDEVAFF1

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented

Accessible at offset 0xFAC from PMU

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.3.20 PMDEVARCH, Performance Monitors Device Architecture register

The PMDEVARCH characteristics are:

Purpose

Identifies the programmers' model architecture of the Performance Monitor component.

Configuration

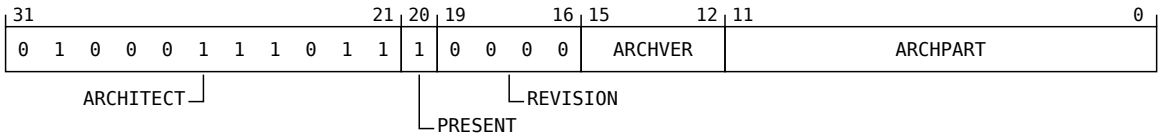
If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is present only when FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMDEVARCH are RES0.

Attributes

PMDEVARCH is a 32-bit register.

Field descriptions



ARCHITECT, bits [31:21]

Defines the architect of the component. For Performance Monitors, this is Arm Limited.

Bits [31:28] are the JEP106 continuation code, `0b0100`.

Bits [27:21] are the JEP106 identification code, `0b0111011`.

Reads as `0b01000111011`

Access to this field is RO.

PRESENT, bit [20]

DEVARCH present. Indicates that the PMDEVARCH register is present.

Reads as `0b1`

Access to this field is RO.

REVISION, bits [19:16]

Defines the architecture revision. For architectures defined by Arm this is the minor revision.

For Performance Monitors, the revision defined by Armv8 is `0x0`.

All other values are reserved.

Reads as `0b0000`

Access to this field is RO.

ARCHVER, bits [15:12]

When UInt(PMU.PMDEVARCH.ARCHPART) == `0xA16`:

Architecture Version. Defines the architecture version of the component.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ARCHVER	Meaning
0b0010	Performance Monitors Extension version 3, PMUv3.

All other values are reserved.

PMDEVARCH.ARCHVER and PMDEVARCH.ARCHPART are also defined as a single field, PMDEVARCH.ARCHID, so that PMDEVARCH.ARCHVER is PMDEVARCH.ARCHID[15:12].

Access to this field is RO.

Otherwise:

Architecture Version. Defines the architecture version of the component.

ARCHVER	Meaning
0b0000	PC Sample-based Profiling version 2, FEAT_PCSRv8p2.

All other values are reserved.

PMDEVARCH.ARCHVER and PMDEVARCH.ARCHPART are also defined as a single field, PMDEVARCH.ARCHID, so that PMDEVARCH.ARCHVER is PMDEVARCH.ARCHID[15:12].

ARCHPART, bits [11:0]

Architecture Part. Defines the architecture of the component.

The value of this field is an IMPLEMENTATION DEFINED choice of:

ARCHPART	Meaning
0xA10	PC Sample-based Profiling, including the 32-bit programmers' model extension.
0xA16	Armv8-A PE performance monitors, including the 32-bit programmers' model extension.

FEAT_PMUv3_EXT32 implements the functionality described by the values 0xA10 and 0xA16.

The values 0xA10 and 0xA20 indicate that FEAT_PCSRv8p2 is implemented but the Performance Monitors Extension is not implemented.

PMDEVARCH.ARCHVER and PMDEVARCH.ARCHPART are also defined as a single field, PMDEVARCH.ARCHID, so that PMDEVARCH.ARCHPART is PMDEVARCH.ARCHID[11:0].

Access to this field is RO.

Accessing PMDEVARCH

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset 0xFBC from PMU

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.3.21 PMDEVID, Performance Monitors Device ID register

The PMDEVID characteristics are:

Purpose

Provides information about features of the Performance Monitors implementation.

Configuration

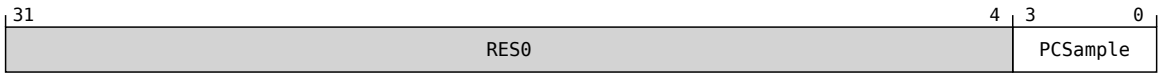
If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is present only when (v8Ap2 or FEAT_PCSRv8p2 is implemented) and FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMDEVID are RES0.

Attributes

PMDEVID is a 32-bit register.

Field descriptions



Bits [31:4]

Reserved, RES0.

PCSample, bits [3:0]

Indicates the level of PC Sample-based Profiling support using Performance Monitors registers.

The value of this field is an IMPLEMENTATION DEFINED choice of:

PCSample	Meaning
0b0000	PC Sample-based Profiling Extension is not implemented in the Performance Monitors register space.
0b0001	PC Sample-based Profiling Extension is implemented in the Performance Monitors register space.

All other values are reserved.

FEAT_PCSRv8p2 implements the functionality identified by the value 0b0001.

If FEAT_PCSRv8p2 is not implemented, then the only permitted value is 0b0000.

From Armv8.2, when FEAT_PCSRv8p2 is implemented, the value 0b0000 is not permitted.

Access to this field is RO.

Accessing PMDEVID

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented and (v8Ap2 or FEAT_PCSRv8p2 is implemented)

Accessible at offset 0xFC8 from PMU

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.3.22 PMDEVTYPE, Performance Monitors Device Type register

The PMDEVTYPE characteristics are:

Purpose

Indicates to a debugger that this component is part of a PE's performance monitor interface.

Configuration

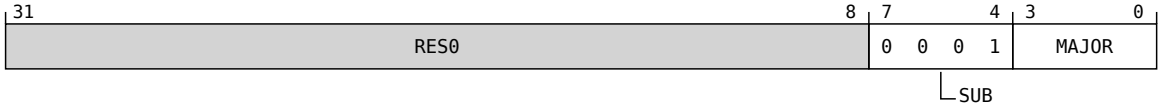
If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is present only when FEAT_PMUv3_EXT is implemented and an implementation implements PMDEVTYPE. Otherwise, direct accesses to PMDEVTYPE are RES0.

Attributes

PMDEVTYPE is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

SUB, bits [7:4]

Subtype. Indicates this is a component within a PE.

Reads as `0b0001`

Access to this field is RO.

MAJOR, bits [3:0]

Major type.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MAJOR	Meaning
`0b0000`	Unspecified.
`0b0110`	Performance monitor component.

FEAT_PMUv3 implements the functionality described by the value `0b0110`.

Access to this field is RO.

Accessing PMDEVTYPE

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset `0xFCC` from PMU

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.3.23 PMEVCNTR<n>_EL0, Performance Monitors Event Count Registers, n = 0 - 30

The PMEVCNTR<n>_EL0 characteristics are:

Purpose

Holds event counter <n>, which counts events, where <n> is 0 to 30.

Configuration

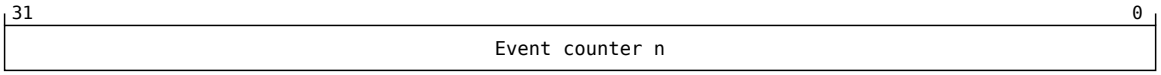
PMEVCNTR<n>_EL0 is in the Core power domain.

This register is present only when FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMEVCNTR<n>_EL0 are RES0.

Attributes

PMEVCNTR<n>_EL0 is a 32-bit register.

Field descriptions



EVCNT, bits [31:0]

Event counter n. Value of event counter n, where n is the number of this register and is a number from 0 to 30.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PMEVCNTR<n>_EL0

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented

[31:0] Accessible at offset `0x000 + (8 * n)` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.3.24 PMEVTYPER<n>_EL0, Performance Monitors Event Type Registers, n = 0 - 30

The PMEVTYPER<n>_EL0 characteristics are:

Purpose

Configures event counter n, where n is 0 to 30.

Configuration

PMEVTYPER<n>_EL0 is in the Core power domain.

If event counter n is not implemented:

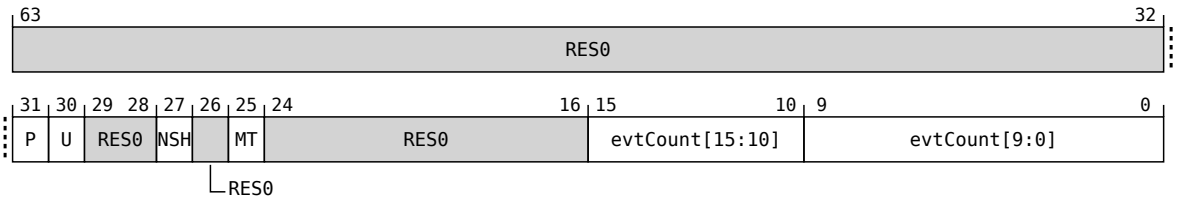
- When IsCorePowered() && !DoubleLockStatus() && !OSLockStatus() && AllowExternalPMUAccess(), accesses are RES0.
- Otherwise, it is CONSTRAINED UNPREDICTABLE whether accesses to this register are RES0 or generate an error response.

This register is present only when FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMEVTYPER<n>_EL0 are RES0.

Attributes

PMEVTYPER<n>_EL0 is a 64-bit register.

Field descriptions



Bits [63:32]

Reserved, RES0.

P, bit [31]

EL1 filtering. Controls counting events in EL1.

P	Meaning
0b0	This mechanism has no effect on filtering of events.
0b1	The PE does not count events in EL1.

If Secure and Non-secure states are implemented, then counting events in Non-secure EL1 is further controlled by PMEVTYPER<n>_EL0.NSK.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

U, bit [30]

EL0 filtering. Controls counting events in EL0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	Meaning
0b0	This mechanism has no effect on filtering of events.
0b1	The PE does not count events in EL0.

If Secure and Non-secure states are implemented, then counting events in Non-secure EL0 is further controlled by PMEVTYPER<n>_EL0.NSU.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [29:28]

Reserved, RES0.

NSH, bit [27]

EL2 filtering. Controls counting events in EL2.

NSH	Meaning
0b0	The PE does not count events in EL2.
0b1	This mechanism has no effect on filtering of events.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bit [26]

Reserved, RES0.

MT, bit [25]

When an IMPLEMENTATION DEFINED multi-threaded PMU extension is implemented:

Multithreading.

MT	Meaning
0b0	Count events only on controlling PE.
0b1	Count events from any PE with the same affinity at level 1 and above as this PE.

Unless otherwise stated:

- If the event counts PE cycles when a stall condition is true and a second condition is true, then the counter counts Processor cycles when the stall condition is true for all of these PEs, and the second condition is true for any of these PEs.
- If the event counts PE cycles when any other condition is true, then the counter counts Processor cycles when the condition is true for any of these PEs.
- Otherwise, the event counts by the sum of the count across all of these PEs.

For the stall events, the stall condition means the applicable condition described by the STALL, STALL_FRONTEND, or STALL_BACKEND event.

The second condition is any condition in addition to this.

For example, for the STALL_FRONTEND_L1I event, the stall condition is STALL_FRONTEND, and the second condition is when there is a demand instruction miss in the first level of instruction cache.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For the STALL, STALL_FRONTEND, and STALL_BACKEND events themselves, the second condition is the null TRUE condition.

See 'Multithreaded implementations' and 'Cycle event counting in multithreaded implementations'.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [24:16]

Reserved, RES0.

evtCount[15:10], bits [15:10]

When FEAT_PMUv3p1 is implemented:

Extension to evtCount[9:0]. For more information, see evtCount[9:0].

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

evtCount[9:0], bits [9:0]

Event to count.

The event number of the event that is counted by event counter PMU.PMEVCNTR<n>_EL0.

The ranges of event numbers allocated to each type of event are shown in 'Allocation of the PMU event number space'.

If PMEVTYPER<n>_EL0.evtCount is programmed to an event that is reserved or not supported by the PE, the behavior depends on the value written:

- For the range `0x0000` to `0x003F`, no events are counted and the value returned by a direct or external read of the PMEVTYPER<n>_EL0.evtCount field is the value written to the field.
- If FEAT_PMUv3p1 is implemented, for the range `0x4000` to `0x403F`, no events are counted and the value returned by a direct or external read of the PMEVTYPER<n>_EL0.evtCount field is the value written to the field.
- For other values, it is UNPREDICTABLE what event, if any, is counted and the value returned by a direct or external read of the PMEVTYPER<n>_EL0.evtCount field is UNKNOWN.

Note

UNPREDICTABLE means the event must not expose privileged information.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PMEVTYPER<n>_EL0

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented

[31:0] Accessible at offset `0x400 + (4 * n)` from PMU

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.3.25 PMEVFILT2R<n>, Performance Monitors Event Filter Registers, n = 0 - 63

The PMEVFILT2R<n> characteristics are:

Purpose

Provides additional IMPLEMENTATION DEFINED configuration controls for PMU counters.

Each PMEVFILT2R<n> register can provide additional configuration controls for a PMU counter, where:

- For values of n less than 31, if event counter n is implemented, then the controls are for PMU event counter <n>.
- For n equal to 31, the controls are for the cycle counter, PMCCNTR_EL0;
- For all other values of n, PMEVFILT2R<n> is not implemented.

Although this mapping is recommended, it is not required and the function of each register is IMPLEMENTATION DEFINED.

Configuration

PMEVFILT2R<n> is in the Core power domain.

If PMEVFILT2R<n> is not implemented:

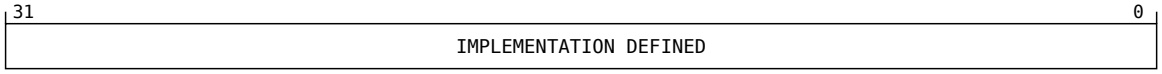
- When IsCorePowered() && !DoubleLockStatus() && !OSLockStatus() && AllowExternalPMUAccess(), accesses are RES0.
- Otherwise, it is CONSTRAINED UNPREDICTABLE whether accesses to this register are RES0 or generate an error response.

This register is present only when FEAT_PMUv3_EXT is implemented and an implementation implements PMEVFILT2R<n>. Otherwise, direct accesses to PMEVFILT2R<n> are RES0.

Attributes

PMEVFILT2R<n> is a 32-bit register.

Field descriptions



IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

Accessing PMEVFILT2R<n>

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented

[31:0] Accessible at offset `0x800 + (4 * n)` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.3.26 PMIIDR, Performance Monitors Implementation Identification Register

The PMIIDR characteristics are:

Purpose

Provides discovery information about the Performance Monitor component.

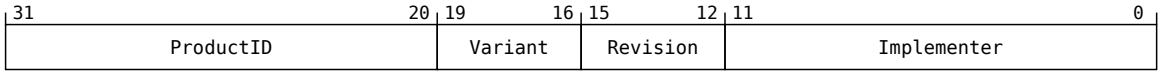
Configuration

This register is present only when FEAT_PMUv3_EXT32 is implemented and an implementation implements PMIIDR. Otherwise, direct accesses to PMIIDR are RES0.

Attributes

PMIIDR is a 32-bit register.

Field descriptions



ProductID, bits [31:20]

Part number, bits [11:0]. The part number is selected by the designer of the component.

Matches the PMPIDR1.PART_1, PMPIDR0.PART_0 fields if present.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Variant, bits [19:16]

Component major revision.

Defines either a variant of the component defined by PMIIDR.ProductID, or the major revision of the component.

When defining a major revision, PMIIDR.Variant and PMIIDR.Revision together form the revision number of the component, with this field being the most significant part.

When a component is changed, PMIIDR.Variant or PMIIDR.Revision is increased to ensure that software can differentiate between different revisions of the component. If this field is increased, PMIIDR.Revision should be set to `0b0000`.

Matches the PMPIDR2.REVISION field, if present.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Revision, bits [15:12]

Component minor revision.

PMIIDR.Variant and PMIIDR.Revision together form the revision number of the component, with this field being the least significant part.

When a component is changed, PMIIDR.Variant or PMIIDR.Revision is increased to ensure that software can differentiate between different revisions of the component. If PMIIDR.Variant field is increased, this field should be set to `0b0000`, otherwise the value in this field should be increased.

Matches the PMPIDR3.REVAND field, if present.

This field has an IMPLEMENTATION DEFINED value.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Access to this field is RO.

Implementer, bits [11:0]

Contains the JEP106 manufacturer's identification code of the designer of the PMU.

The code identifies the designer of the component, which might not be the same as the implementer of the device containing the component.

Zero is not a valid JEP106 identification code, meaning a value of zero for PMIIDR indicates this register is not implemented.

For an implementation designed by Arm, this field reads as `0x43B`.

This field has an IMPLEMENTATION DEFINED value.

Bits [11:8] contain the JEP106 bank identifier of the designer minus 1.

Bit 7 is RES0.

Bits [6:0] contain bits [6:0] of the JEP106 manufacturer's identification code of the designer.

If PMPIDR4 is implemented, PMPIDR4.DES_2 matches bits [11:8] of this field.

If PMPIDR2 is implemented, PMPIDR2.DES_1 matches bits [6:4] of this field.

If PMPIDR1 is implemented, PMPIDR1.DES_0 matches bits [3:0] of this field.

Access to this field is RO.

Accessing PMIIDR

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset `0xE08` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.3.27 PMINTENCLR_EL1, Performance Monitors Interrupt Enable Clear Register

The PMINTENCLR_EL1 characteristics are:

Purpose

Allows software to disable the generation of interrupt requests on overflows from the following counters:

- The cycle counter PMCCNTR_EL0.
- The event counters PMEVCNTR<n>_EL0.

Reading from this register shows which overflow interrupt requests are enabled.

Configuration

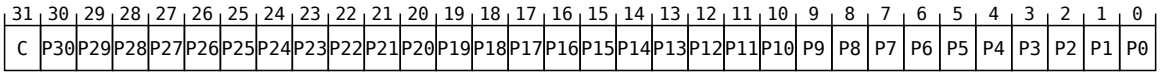
PMINTENCLR_EL1 is in the Core power domain.

This register is present only when FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMINTENCLR_EL1 are RES0.

Attributes

PMINTENCLR_EL1 is a 32-bit register.

Field descriptions



C, bit [31]

Interrupt request on unsigned overflow of PMCCNTR_EL0 disable. On writes, allows software to disable the interrupt request on unsigned overflow of PMCCNTR_EL0. On reads, returns the interrupt request on unsigned overflow of PMCCNTR_EL0 enable status.

C	Meaning
0b0	Interrupt request on unsigned overflow of PMCCNTR_EL0 disabled.
0b1	Interrupt request on unsigned overflow of PMCCNTR_EL0 enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is W1C.

P<m> , bits [m], for m = 30 to 0

Interrupt request on unsigned overflow of PMEVCNTR<m>_EL0 disable. On writes, allows software to disable the interrupt request on unsigned overflow of PMEVCNTR<m>_EL0. On reads, returns the interrupt request on unsigned overflow of PMEVCNTR<m>_EL0 enable status.

P<m>	Meaning
0b0	Interrupt request on unsigned overflow of PMEVCNTR<m>_EL0 disabled.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b1	Interrupt request on unsigned overflow of PMEVCNTR<m>_EL0 enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= NUM_PMU_COUNTERS, access to this field is RAZ/WI.
- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is W1C.

Accessing PMINTENCLR_EL1

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented

[31:0] Accessible at offset 0xC60 from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.3.28 PMINTENSET_EL1, Performance Monitors Interrupt Enable Set Register

The PMINTENSET_EL1 characteristics are:

Purpose

Allows software to enable the generation of interrupt requests on overflows from the following counters:

- The cycle counter PMCCNTR_EL0.
- The event counters PMEVCNTR<n>_EL0.

Reading from this register shows which overflow interrupt requests are enabled.

Configuration

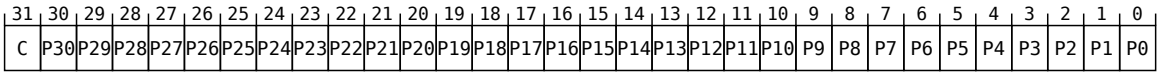
PMINTENSET_EL1 is in the Core power domain.

This register is present only when FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMINTENSET_EL1 are RES0.

Attributes

PMINTENSET_EL1 is a 32-bit register.

Field descriptions



C, bit [31]

Interrupt request on unsigned overflow of PMCCNTR_EL0 enable. On writes, allows software to enable the interrupt request on unsigned overflow of PMCCNTR_EL0. On reads, returns the interrupt request on unsigned overflow of PMCCNTR_EL0 enable status.

C	Meaning
0b0	Interrupt request on unsigned overflow of PMCCNTR_EL0 disabled.
0b1	Interrupt request on unsigned overflow of PMCCNTR_EL0 enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is W1S.

P<m> , bits [m], for m = 30 to 0

Interrupt request on unsigned overflow of PMEVCNTR<m>_EL0 enable. On writes, allows software to enable the interrupt request on unsigned overflow of PMEVCNTR<m>_EL0. On reads, returns the interrupt request on unsigned overflow of PMEVCNTR<m>_EL0 enable status.

P<m>	Meaning
0b0	Interrupt request on unsigned overflow of PMEVCNTR<m>_EL0 disabled.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b1	Interrupt request on unsigned overflow of PMEVCNTR<m>_EL0 enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= NUM_PMU_COUNTERS, access to this field is RAZ/WI.
- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is W1S.

Accessing PMINTENSET_EL1

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented

[31:0] Accessible at offset 0xC40 from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.3.29 PMITCTRL, Performance Monitors Integration mode Control register

The PMITCTRL characteristics are:

Purpose

Enables the Performance Monitors to switch from default mode into integration mode, where test software can control directly the inputs and outputs of the PE, for integration testing or topology detection.

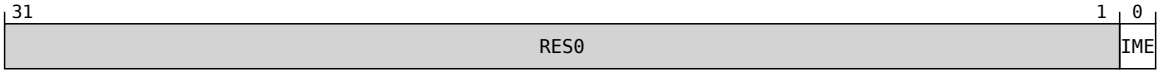
Configuration

This register is present only when FEAT_PMUv3_EXT is implemented and an implementation implements PMITCTRL. Otherwise, direct accesses to PMITCTRL are RES0.

Attributes

PMITCTRL is a 32-bit register.

Field descriptions



Bits [31:1]

Reserved, RES0.

IME, bit [0]

Integration mode enable. When IME == 1, the device reverts to an integration mode to enable integration testing or topology detection.

IME	Meaning
0b0	Normal operation.
0b1	Integration mode enabled.

The integration mode behavior is IMPLEMENTATION DEFINED.

The following resets apply:

- If the register is implemented in the Core power domain:
  - On a Cold reset, this field resets to 0.
  - On an External debug reset, the value of this field is unchanged.
  - On a Warm reset, the value of this field is unchanged.
- If the register is implemented in the External debug power domain:
  - On a Cold reset, the value of this field is unchanged.
  - On an External debug reset, this field resets to 0.
  - On a Warm reset, the value of this field is unchanged.

Accessing PMITCTRL

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset 0xF00 from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.3.30 PMLAR, Performance Monitors Lock Access Register

The PMLAR characteristics are:

Purpose

Allows or disallows access to the Performance Monitors registers through a memory-mapped interface.

The optional Software Lock provides a lock to prevent memory-mapped writes to the Performance Monitors registers. Use of this lock mechanism reduces the risk of accidental damage to the contents of the Performance Monitors registers. It does not, and cannot, prevent all accidental or malicious damage.

Configuration

If FEAT_DoPD is implemented, Software Lock is not implemented by the architecturally-defined debug components of the PE in the Core power domain.

If FEAT_DoPD is not implemented, this register is in the Debug power domain.

Software uses PMLAR to set or clear the lock, and PMLSR to check the current status of the lock.

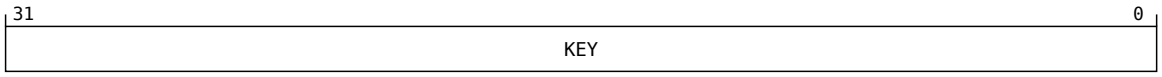
This register is present only when FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMLAR are RES0.

Attributes

PMLAR is a 32-bit register.

Field descriptions

When PMU Software Lock is implemented:

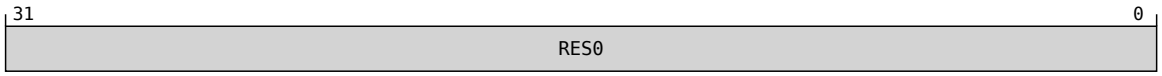


KEY, bits [31:0]

Lock Access control. Writing the key value 0xC5ACCE55 to this field unlocks the lock, enabling write accesses to this component's registers through a memory-mapped interface.

Writing any other value to this register locks the lock, disabling write accesses to this component's registers through a memory mapped interface.

Otherwise:



Otherwise

Bits [31:0]

Reserved, RES0.

Accessing PMLAR

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset 0xFB0 from PMU

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are WO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.3.31 PMLSR, Performance Monitors Lock Status Register

The PMLSR characteristics are:

Purpose

Indicates the current status of the software lock for Performance Monitors registers.

The optional Software Lock provides a lock to prevent memory-mapped writes to the Performance Monitors registers. Use of this lock mechanism reduces the risk of accidental damage to the contents of the Performance Monitors registers. It does not, and cannot, prevent all accidental or malicious damage.

Configuration

If FEAT_DoPD is implemented, Software Lock is not implemented by the architecturally-defined debug components of the PE in the Core power domain.

If FEAT_DoPD is not implemented, this register is in the Debug power domain.

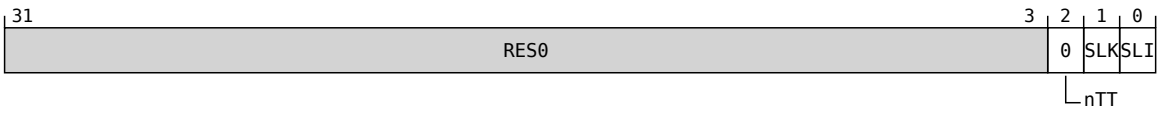
Software uses PMLAR to set or clear the lock, and PMLSR to check the current status of the lock.

This register is present only when FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMLSR are RES0.

Attributes

PMLSR is a 32-bit register.

Field descriptions



Bits [31:3]

Reserved, RES0.

nTT, bit [2]

Not thirty-two bit access required.

Reads as `0b0`

Access to this field is RO.

SLK, bit [1]

When PMU Software Lock is implemented and FEAT_DoPD is not implemented:

Software Lock status for this component. For an access to LSR that is not a memory-mapped access, or when Software Lock is not implemented, this field is RES0.

For memory-mapped accesses when Software Lock is implemented, possible values of this field are:

SLK	Meaning
`0b0`	Lock clear. Writes are permitted to this component's registers.
`0b1`	Lock set. Writes to this component's registers are ignored, and reads have no side effects.

The reset behavior of this field is:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a External debug reset, this field resets to '1'.

Otherwise:

Reserved, RAZ.

SLI, bit [0]

Software Lock implemented. For an access to LSR that is not a memory-mapped access, this field is RAZ.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SLI	Meaning
0b0	Software Lock not implemented or not memory-mapped access.
0b1	Software Lock implemented and memory-mapped access.

Access to this field is RO.

Accessing PMLSR

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset 0xFB4 from PMU

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.3.32 PMMIR, Performance Monitors Machine Identification Register

The PMMIR characteristics are:

Purpose

Describes Performance Monitors parameters specific to the implementation.

Configuration

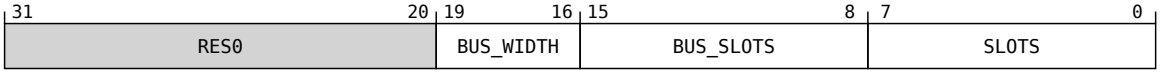
PMMIR is in the Core power domain.

This register is present only when FEAT_PMUv3_EXT is implemented and FEAT_PMUv3p4 is implemented. Otherwise, direct accesses to PMMIR are RES0.

Attributes

PMMIR is a 32-bit register.

Field descriptions



Bits [31:20]

Reserved, RES0.

BUS_WIDTH, bits [19:16]

Bus width. Indicates the number of bytes each BUS_ACCESS event relates to. Encoded as Log_2(number of bytes), plus one.

The value of this field is an IMPLEMENTATION DEFINED choice of:

BUS_WIDTH	Meaning
0b0000	The information is not available.
0b0011	Four bytes.
0b0100	8 bytes.
0b0101	16 bytes.
0b0110	32 bytes.
0b0111	64 bytes.
0b1000	128 bytes.
0b1001	256 bytes.
0b1010	512 bytes.
0b1011	1024 bytes.
0b1100	2048 bytes.

All other values are reserved.

Each transfer is up to this number of bytes. An access might be smaller than the bus width.

When this field is nonzero, each access counted by BUS_ACCESS is at most BUS_WIDTH bytes. An implementation might treat a wide bus as multiple narrower buses, such that a wide access on the bus increments the BUS_ACCESS counter by more than one.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Access to this field is RO.

BUS_SLOTS, bits [15:8]

Bus count. The largest value by which the BUS_ACCESS event might increment in a single BUS_CYCLES cycle.

This field has an IMPLEMENTATION DEFINED value.

When this field is nonzero, the largest value by which the BUS_ACCESS event might increment in a single BUS_CYCLES cycle is BUS_SLOTS.

If the bus count information is not available, this field will read as zero.

Access to this field is RO.

SLOTS, bits [7:0]

Operation width. The largest value by which the STALL_SLOT event might increment in a single cycle. If the STALL_SLOT event is not implemented, this field might read as zero.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing PMMIR

If the Core power domain is off or in a low-power state, access to this register returns an Error.

Accesses to this register use the following encodings:

When FEAT_PMUv3p4 is implemented and FEAT_PMUv3_EXT32 is implemented

[31:0] Accessible at offset `0xE40` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.3.33 PMOVSCLR_EL0, Performance Monitors Overflow Flag Status Clear register

The PMOVSCLR_EL0 characteristics are:

Purpose

Allows software to clear the unsigned overflow flags for the following counters to 0:

- The cycle counter PMCCNTR_EL0.
- The event counters PMEVCNTR<n>_EL0.

Reading from this register shows the current unsigned overflow flag values.

Configuration

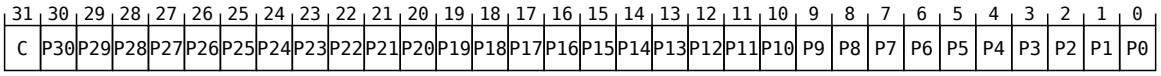
PMOVSCLR_EL0 is in the Core power domain.

This register is present only when FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMOVSCLR_EL0 are RES0.

Attributes

PMOVSCLR_EL0 is a 32-bit register.

Field descriptions



C, bit [31]

Unsigned overflow flag for PMCCNTR_EL0 clear. On writes, allows software to clear the unsigned overflow flag for PMCCNTR_EL0 to 0. On reads, returns the unsigned overflow flag for PMCCNTR_EL0 overflow status.

C	Meaning
0b0	PMCCNTR_EL0 has not overflowed.
0b1	PMCCNTR_EL0 has overflowed.

PMCR_EL0.LC controls whether an overflow is detected from unsigned overflow of PMCCNTR_EL0[31:0] or unsigned overflow of PMCCNTR_EL0[63:0].

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is W1C.

P<m> , bits [m], for m = 30 to 0

Unsigned overflow flag for PMEVCNTR<m>_EL0 clear. On writes, allows software to clear the unsigned overflow flag for PMEVCNTR<m>_EL0 to 0. On reads, returns the unsigned overflow flag for PMEVCNTR<m>_EL0 overflow status.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

P<m>	Meaning
0b0	PMEVCNTR<m>_EL0 has not overflowed.
0b1	PMEVCNTR<m>_EL0 has overflowed.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= NUM_PMU_COUNTERS, access to this field is RAZ/WI.
- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is W1C.

Accessing PMOVSCLR_EL0

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented

[31:0] Accessible at offset 0xC80 from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.3.34 PMOVSSET_EL0, Performance Monitors Overflow Flag Status Set Register

The PMOVSSET_EL0 characteristics are:

Purpose

Allows software to set the unsigned overflow flags for the following counters to 1:

- The cycle counter PMCCNTR_EL0.
- The event counters PMEVCNTR<n>_EL0.

Reading from this register shows the current unsigned overflow flag values.

Configuration

PMOVSSET_EL0 is in the Core power domain.

This register is present only when FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMOVSSET_EL0 are RES0.

Attributes

PMOVSSET_EL0 is a 32-bit register.

Field descriptions

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

C, bit [31]

Unsigned overflow flag for PMCCNTR_EL0 set. On writes, allows software to set the unsigned overflow flag for PMCCNTR_EL0 to 1. On reads, returns the unsigned overflow flag for PMCCNTR_EL0 overflow status.

C	Meaning
0b0	PMCCNTR_EL0 has not overflowed.
0b1	PMCCNTR_EL0 has overflowed.

PMCR_EL0.LC controls whether an overflow is detected from unsigned overflow of PMCCNTR_EL0[31:0] or unsigned overflow of PMCCNTR_EL0[63:0].

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is W1S.

P<m> , bits [m], for m = 30 to 0

Unsigned overflow flag for PMEVCNTR<m>_EL0 set. On writes, allows software to set the unsigned overflow flag for PMEVCNTR<m>_EL0 to 1. On reads, returns the unsigned overflow flag for PMEVCNTR<m>_EL0 overflow status.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

P<m>	Meaning
0b0	PMEVCNTR<m>_EL0 has not overflowed.
0b1	PMEVCNTR<m>_EL0 has overflowed.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= NUM_PMU_COUNTERS, access to this field is RAZ/WI.
- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is W1S.

Accessing PMOVSSET_EL0

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented

[31:0] Accessible at offset 0xCC0 from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.3.35 PMPCSR, Program Counter Sample Register

The PMPCSR characteristics are:

Purpose

Holds a sampled instruction address value.

Configuration

PMPCSR is in the Core power domain.

This register is a PC Sample-based Profiling Extension register.

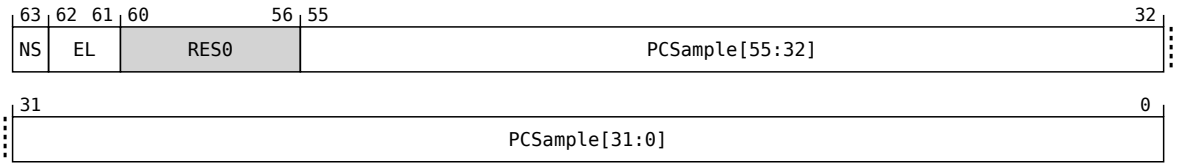
If FEAT_PMUv3_EXT32 is implemented, support for 64-bit atomic reads is IMPLEMENTATION DEFINED. If 64-bit atomic reads are implemented, a 64-bit read of PMPCSR has the same side-effect as a 32-bit read of PMCSR[31:0] followed by a 32-bit read of PMPCSR[63:32], returning the combined value. For example, if the PE is in Debug state, then a 64-bit atomic read returns bits[31:0] == 0xFFFFFFFF and bits[63:32] UNKNOWN.

This register is present only when FEAT_PMUv3_EXT is implemented and FEAT_PCSRv8p2 is implemented. Otherwise, direct accesses to PMPCSR are RES0.

Attributes

PMPCSR is a 64-bit register.

Field descriptions



NS, bit [63]

Non-secure state sample. Indicates the Security state that is associated with the most recent PMPCSR sample or, when it is read as a single atomic 64-bit read, the current PMPCSR sample.

If EL3 is not implemented, this bit indicates the Effective value of SCR.NS.

NS	Meaning
0b0	Sample is from Secure state.
0b1	Sample is from Non-secure state.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

EL, bits [62:61]

Exception level status sample. Indicates the Exception level that is associated with the most recent PMPCSR sample or, when it is read as a single atomic 64-bit read, the current PMPCSR sample.

EL	Meaning
0b00	Sample is from EL0.
0b01	Sample is from EL1.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EL	Meaning
0b10	Sample is from EL2.
0b11	Sample is from EL3.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [60:56]

Reserved, RES0.

PCSample[55:32], bits [55:32]

Bits[55:32] of the sampled instruction address value. The translation regime that PMPCSR samples can be determined from PMPCSR.{NS,EL}.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

PCSample[31:0], bits [31:0]

Bits[31:0] of the sampled instruction address value.

PMPCSR[31:0] reads as 0xFFFFFFFF when any of the following are true:

- The PE is in Debug state.
- PC Sample-based profiling is prohibited.

If a branch instruction has retired since the PE left reset state, then the first read of PMPCSR[31:0] is permitted but not required to return 0xFFFFFFFF.

PMPCSR[31:0] reads as an UNKNOWN value when any of the following are true:

- The PE is in reset state.
- No branch instruction has retired since the PE left reset state, Debug state, or a state where PC Sample-based Profiling is prohibited.
- No branch instruction has retired since the last read of PMPCSR[31:0].

For the cases where a read of PMPCSR[31:0] returns 0xFFFFFFFF or an UNKNOWN value, the read has the side-effect of setting PMPCSR[63:32], PMCID1SR, PMCID2SR, and PMVIDSR to UNKNOWN values.

Otherwise, a read of PMPCSR[31:0] returns bits [31:0] of the sampled instruction address value and has the side-effect of indirectly writing to PMPCSR[63:32], PMCID1SR, PMCID2SR, and PMVIDSR. The translation regime that PMPCSR samples can be determined from PMPCSR.{NS,EL}.

For a read of PMPCSR[31:0] from the memory-mapped interface, if PMLSR.SLK == 1, meaning the OPTIONAL Software Lock is locked, then the side-effect of the access does not occur and PMPCSR[63:32], PMCID1SR, PMCID2SR, and PMVIDSR are unchanged.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing PMPCSR

IMPLEMENTATION DEFINED extensions to external debug might make the value of this register UNKNOWN, see 'Permitted behavior that might make the PC Sample-based profiling registers UNKNOWN'.

Note

A 32-bit access to PMPCSR[63:32] does not update the PC sample registers. Only a 64-bit access to PMPCSR[63:0] or a 32-bit access to PMPCSR[31:0] updates the PC sample registers. This includes the value a subsequent 32-bit read of PMPCSR[63:32] will return.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented and FEAT_PCSRv8p2 is implemented

[31:0] Accessible at offset `0x200` from PMU

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register generate an error response.
- When OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

When FEAT_PMUv3_EXT32 is implemented and FEAT_PCSRv8p2 is implemented

[63:32] Accessible at offset `0x204` from PMU

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register generate an error response.
- When OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

When FEAT_PMUv3_EXT32 is implemented and FEAT_PCSRv8p2 is implemented

[31:0] Accessible at offset `0x220` from PMU

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register generate an error response.
- When OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

When FEAT_PMUv3_EXT32 is implemented and FEAT_PCSRv8p2 is implemented

[63:32] Accessible at offset `0x224` from PMU

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register generate an error response.
- When OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.3.36 PMPIDR0, Performance Monitors Peripheral Identification Register 0

The PMPIDR0 characteristics are:

Purpose

Provides information to identify a Performance Monitor component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

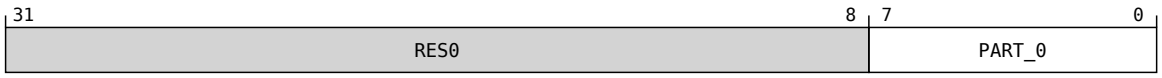
This register is required for CoreSight compliance.

This register is present only when FEAT_PMUv3_EXT is implemented and an implementation implements PMPIDR0. Otherwise, direct accesses to PMPIDR0 are RES0.

Attributes

PMPIDR0 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

PART_0, bits [7:0]

Part number, least significant byte.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing PMPIDR0

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset `0xFE0` from PMU

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.3.37 PMPIDR1, Performance Monitors Peripheral Identification Register 1

The PMPIDR1 characteristics are:

Purpose

Provides information to identify a Performance Monitor component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

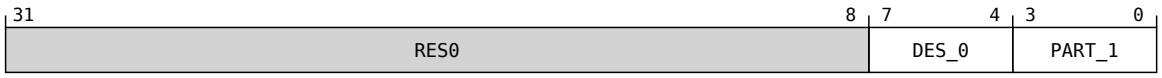
This register is required for CoreSight compliance.

This register is present only when FEAT_PMUv3_EXT is implemented and an implementation implements PMPIDR1. Otherwise, direct accesses to PMPIDR1 are RES0.

Attributes

PMPIDR1 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

DES_0, bits [7:4]

Designer, least significant nibble of JEP106 ID code. For Arm Limited, this field is `0b1011`.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

PART_1, bits [3:0]

Part number, most significant nibble.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing PMPIDR1

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset `0xFE4` from PMU

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.3.38 PMPIDR2, Performance Monitors Peripheral Identification Register 2

The PMPIDR2 characteristics are:

Purpose

Provides information to identify a Performance Monitor component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

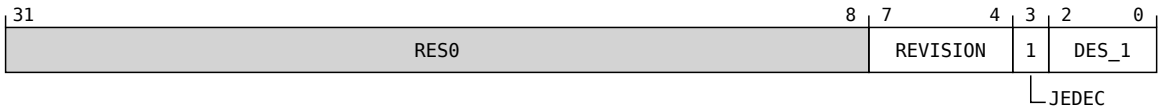
This register is required for CoreSight compliance.

This register is present only when FEAT_PMUv3_EXT is implemented and an implementation implements PMPIDR2. Otherwise, direct accesses to PMPIDR2 are RES0.

Attributes

PMPIDR2 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

REVISION, bits [7:4]

Part major revision. Parts can also use this field to extend Part number to 16-bits.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

JEDEC, bit [3]

Indicates a JEP106 identity code is used.

Reads as `0b1`

Access to this field is RO.

DES_1, bits [2:0]

Designer, most significant bits of JEP106 ID code. For Arm Limited, this field is `0b011`.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing PMPIDR2

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset `0xFE8` from PMU

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.3.39 PMPIDR3, Performance Monitors Peripheral Identification Register 3

The PMPIDR3 characteristics are:

Purpose

Provides information to identify a Performance Monitor component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

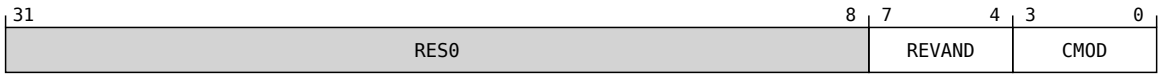
This register is required for CoreSight compliance.

This register is present only when FEAT_PMUv3_EXT is implemented and an implementation implements PMPIDR3. Otherwise, direct accesses to PMPIDR3 are RES0.

Attributes

PMPIDR3 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

REVAND, bits [7:4]

Part minor revision. Parts using PMPIDR2.REVISION as an extension to the Part number must use this field as a major revision number.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

CMOD, bits [3:0]

Customer modified. Indicates someone other than the Designer has modified the component.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing PMPIDR3

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset 0xFEC from PMU

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.3.40 PMPIDR4, Performance Monitors Peripheral Identification Register 4

The PMPIDR4 characteristics are:

Purpose

Provides information to identify a Performance Monitor component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

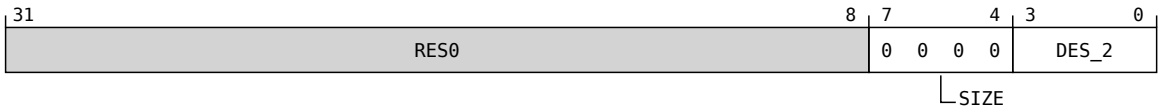
This register is required for CoreSight compliance.

This register is present only when FEAT_PMUv3_EXT is implemented and an implementation implements PMPIDR4. Otherwise, direct accesses to PMPIDR4 are RES0.

Attributes

PMPIDR4 is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

SIZE, bits [7:4]

Size of the component. Log_2 of the number of 4KB pages from the start of the component to the end of the component ID registers.

Reads as `0b0000`

Access to this field is RO.

DES_2, bits [3:0]

Designer, JEP106 continuation code, least significant nibble. For Arm Limited, this field is `0b0100`.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing PMPIDR4

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset `0xFD0` from PMU

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.3.41 PMSWINC_EL0, Performance Monitors Software Increment Register

The PMSWINC_EL0 characteristics are:

Purpose

Increments a counter that is configured to count the Software increment event, event `0x00`. For more information, see 'SW_INCR'.

Configuration

PMSWINC_EL0 is in the Core power domain.

If this register is implemented, use of it is deprecated.

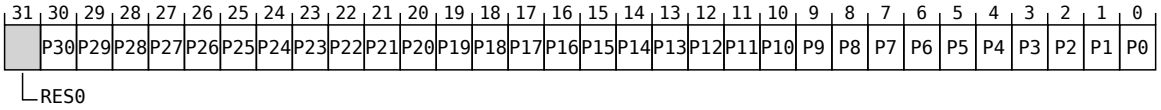
If 1 is written to bit [n] from the external debug interface, it is CONSTRAINED UNPREDICTABLE whether or not a SW_INCR event is created for counter n. This is consistent with not implementing the register in the external debug interface.

This register is present only when FEAT_PMUv3_EXT32 is implemented and an implementation implements PMSWINC_EL0.

Attributes

PMSWINC_EL0 is a 32-bit register.

Field descriptions



Bit [31]

Reserved, RES0.

P<m> , bits [m], for m = 30 to 0

Event counter software increment bit for PMEVCNTR<m>_EL0.

P<m>	Meaning
`0b0`	No action. The write to this bit is ignored.
`0b1`	It is CONSTRAINED UNPREDICTABLE whether a SW_INCR event is generated for event counter m.

Accessing this field has the following behavior:

- When m >= NUM_PMU_COUNTERS, access to this field is RAZ/WI.
- When SoftwareLockStatus(), access to this field is RAZ/WI.
- Otherwise, access to this field is WO/RAZ.

Accessing PMSWINC_EL0

If FEAT_PMUv3p9 or FEAT_PMUv3_EXT64 are implemented, this location is used for accesses to PMZR_EL0.

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

Accesses to this register use the following encodings:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_PMUv3_EXT32 is implemented

Accessible at offset `0xCA0` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(), accesses to this register are WI.
- Otherwise, accesses to this register are WO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.3.42 PMVIDSR, VMID Sample Register

The PMVIDSR characteristics are:

Purpose

Contains the sampled VMID value that is captured on reading PMPCSR[31:0].

Configuration

PMVIDSR is in the Core power domain.

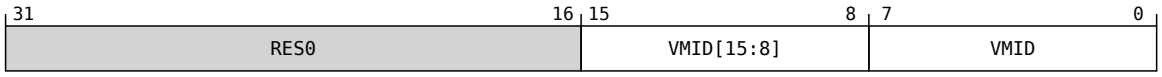
This register is a PC Sample-based Profiling Extension register.

This register is present only when FEAT_PMUv3_EXT32 is implemented and FEAT_PCSRv8p2 is implemented.

Attributes

PMVIDSR is a 32-bit register.

Field descriptions



Bits [31:16]

Reserved, RES0.

VMID[15:8], bits [15:8]

When FEAT_VMID16 is implemented:

Extension to VMID[7:0]. For more information, see VMID[7:0].

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

VMID, bits [7:0]

VMID sample. The VMID associated with the most recent PMPCSR sample. When the most recent PMPCSR sample was generated:

- This field is set to an UNKNOWN value if any of the following apply:
  - The PE is executing at EL2.
- Otherwise:
  - If EL2 is using AArch64 and either FEAT_VMID16 is not implemented or VTCR_EL2.VS is 1, this field is set to VSCTLR_EL2.VMID.
  - If EL2 is using AArch64, FEAT_VMID16 is implemented, and VTCR_EL2.VS is 0, PMVIDSR.VMID[7:0] is set to VSCTLR_EL2.VMID[7:0] and PMVIDSR.VMID[15:8] is RES0.

Because the value written to PMVIDSR is an indirect read of the VMID value, it is CONSTRAINED UNPREDICTABLE whether PMVIDSR is set to the original or new value if PMPCSR samples:

- An instruction that writes to the VMID value.
- The next Context synchronization event.
- Any instruction executed between these two instructions.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing PMVIDSR

If FEAT_PMUv3_EXT64 is implemented, then the same content is present in the same location, and can be accessed using PMVCIDSR[63:32].

IMPLEMENTATION DEFINED extensions to external debug might make the value of this register UNKNOWN, see 'Permitted behavior that might make the PC Sample-based profiling registers UNKNOWN'.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented and FEAT_PCSRv8p2 is implemented

Accessible at offset `0x20C` from PMU

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register generate an error response.
- When OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.4 Generic Timer memory-mapped registers overview

The Generic Timer memory-mapped registers are implemented as multiple register frames, with each register frame having its own base address, as follows:

- A single CNTCTLBase register frame, at base address CNTCTLBase.
- Between one and seven CNTBase*N* register frames, each with its own base address CNTBase*N*.
- For each CNTBase*N* register frame, if required, a CNTEL0Base*N* register frame, at base address CNTEL0Base*N*, that provides an EL0 view of the CNTBase*N* register frame.

For more information, see:

- Memory-mapped timer components.
- The CNTBaseN and CNTEL0BaseN frames. This section includes the memory map of the CNTBase*N* and CNTBase*N* register frames.
- The CNTCTLBase frame. This section includes the memory map of the CNTCTLBase register frame.

Note

Providing a complete set of features in a system level implementation gives an implementation example for a system level implementation of the Generic Timer.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.5 Generic Timer memory-mapped register descriptions

This section describes the Generic Timer registers. Generic Timer memory-mapped registers overview gives an overview of these registers, and includes links to their memory maps.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.5.1 CNTACR<n>, Counter-timer Access Control Registers, n = 0 - 7

The CNTACR<n> characteristics are:

Purpose

Provides top-level access controls for the elements of a timer frame. CNTACR<n> provides the controls for frame CNTBaseN.

In addition to the CNTACR<n> control:

- CNTNSAR controls whether CNTACR<n> is accessible by Non-secure accesses.
- If frame CNTEL0BaseN is implemented, the CNTEL0ACR in frame CNTBaseN provides additional control of accesses to frame CNTEL0BaseN.

Configuration

It is IMPLEMENTATION DEFINED whether CNTACR<n> is implemented in the Core power domain or in the Debug power domain

For more information, see 'Power and reset domains for the system level implementation of the Generic Timer'.

Implemented only if the value of CNTTIDR.Frame<n> is 1.

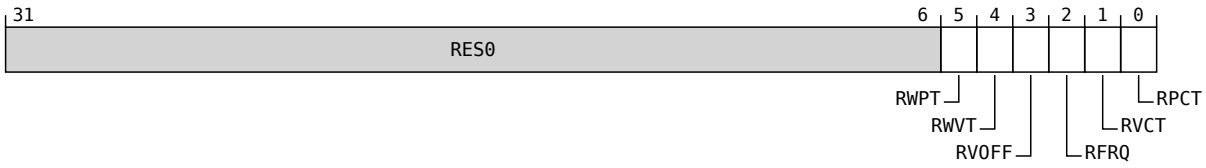
An implementation of the counters might not provide configurable access to some or all of the features. In this case, the associated field in the CNTACR<n> register is:

- RAZ/WI if access is always denied.
- RAO/WI if access is always permitted.

Attributes

CNTACR<n> is a 32-bit register.

Field descriptions



Bits [31:6]

Reserved, RES0.

RWPT, bit [5]

Read/write access to the EL1 Physical Timer registers CNTP_CVAL, CNTP_TVAL, and CNTP_CTL, in frame <n>.

RWPT	Meaning
0b0	No access to the EL1 Physical Timer registers in frame <n>. The registers are RES0.
0b1	Read/write access to the EL1 Physical Timer registers in frame <n>.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

RWVT, bit [4]

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Read/write access to the Virtual Timer register CNTV_CVAL, CNTV_TVAL, and CNTV_CTL, in frame <n>.

RWVT	Meaning
0b0	No access to the Virtual Timer registers in frame <n>. The registers are RES0.
0b1	Read/write access to the Virtual Timer registers in frame <n>.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

RVOFF, bit [3]

Read-only access to CNTVOFF, in frame <n>.

RVOFF	Meaning
0b0	No access to CNTVOFF in frame <n>. The register is RES0.
0b1	Read-only access to CNTVOFF in frame <n>.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

RFRQ, bit [2]

Read-only access to CNTFRQ, in frame <n>.

RFRQ	Meaning
0b0	No access to CNTFRQ in frame <n>. The register is RES0.
0b1	Read-only access to CNTFRQ in frame <n>.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

RVCT, bit [1]

Read-only access to CNTVCT, in frame <n>.

RVCT	Meaning
0b0	No access to CNTVCT in frame <n>. The register is RES0.
0b1	Read-only access to CNTVCT in frame <n>.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

RPCT, bit [0]

Read-only access to CNTPCT, in frame <n>.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RPCT	Meaning
0b0	No access to CNTPCT in frame <n>. The register is RES0.
0b1	Read-only access to CNTPCT in frame <n>.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTACR<n>

In a system that recognizes two Security states:

- CNTACR<n> is always accessible by Secure accesses.
- CNTNSAR.NS<n> determines whether CNTACR<n> is accessible by Non-secure accesses.

CNTACR<n> can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTCTLBase	0x040 + (4 * n)	CNTACR<n>

Accesses to this register are RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.5.2 CNTCR, Counter Control Register

The CNTCR characteristics are:

Purpose

Enables the counter, controls the counter resolution, and controls counter behavior during debug.

Configuration

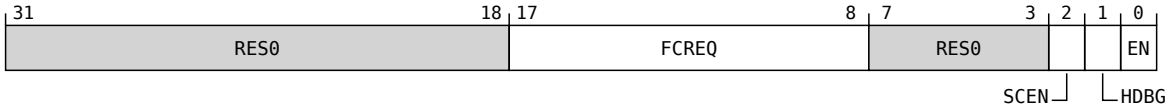
It is IMPLEMENTATION DEFINED whether CNTCR is implemented in the Core power domain or in the Debug power domain

For more information, see 'Power and reset domains for the system level implementation of the Generic Timer'.

Attributes

CNTCR is a 32-bit register.

Field descriptions



Bits [31:18]

Reserved, RES0.

FCREQ, bits [17:8]

Frequency change request. Indicates the number of the entry in the Frequency modes table to select.

Selecting an unimplemented entry, or an entry that contains 0, has no effect on the counter.

The maximum number of entries in the Frequency modes table is IMPLEMENTATION DEFINED up to a maximum of 1004 entries, see 'The Frequency modes table'. An implementation is only required to implement an FCREQ field that can hold values from 0 to the highest supported Frequency modes table entry. Any unrequired most-significant bits of FCREQ can be implemented as RES0.

The reset behavior of this field is:

- On a Timer reset, this field resets to `'0000000000'`.

Bits [7:3]

Reserved, RES0.

SCEN, bit [2]

When FEAT_CNTSC is implemented:

Scale Enable.

SCEN	Meaning
`0b0`	Scaling is not enabled. The counter value is incremented by `0x1`.0000000 for each counter tick.
`0b1`	Scaling is enabled. The counter is incremented by CNTSCR.ScaleVal for each counter tick.

The SCEN bit can only be changed when the counter is disabled, when CNTCR.EN == 0.

If the value of CNTCR.SCEN changes when CNTCR.EN == 1, then:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The counter value becomes UNKNOWN.
- The counter value remains UNKNOWN on future ticks of the clock.

When the CNTCV register in the CNTControlBase frame of the memory mapped counter module is written to, the accumulated fraction information is reset to zero.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HDBG, bit [1]

Halt-on-debug. Controls whether a Halt-on-debug signal halts the system counter:

HDBG	Meaning
0b0	System counter ignores Halt-on-debug.
0b1	Asserted Halt-on-debug signal halts system counter update.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

EN, bit [0]

Enables the counter:

EN	Meaning
0b0	System counter disabled.
0b1	System counter enabled.

The reset behavior of this field is:

- On a Timer reset, this field resets to '0'.

Accessing CNTCR

In a system that supports Secure and Non-secure physical address spaces, the CNTControlBase frame, which includes this register, is implemented only in the Secure physical address space.

CNTCR can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTControlBase	0x000	CNTCR

Accesses to this register are RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.5.3 CNTCV, Counter Count Value register

The CNTCV characteristics are:

Purpose

Indicates the current count value.

Configuration

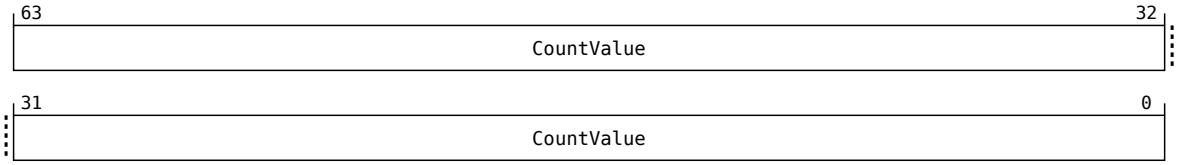
It is IMPLEMENTATION DEFINED whether CNTCV is implemented in the Core power domain or in the Debug power domain

For more information, see 'Power and reset domains for the system level implementation of the Generic Timer'.

Attributes

CNTCV is a 64-bit register.

Field descriptions



CountValue, bits [63:0]

Indicates the counter value.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTCV

Frame	Accessibility
CNTControlBase	RW
CNTReadBase	RO

A write to CNTCV must be visible in the CNTPCT register of each running processor in a finite time.

For the instance of the register in the CNTControlBase frame:

- In a system that supports Secure and Non-secure physical address spaces, this register is implemented only in the Secure physical address space.
- If the counter is enabled, the effect of writing to the register is UNKNOWN.

For the instance of the register in the CNTReadBase frame, this register is accessible in all physical address spaces.

In an implementation that supports 64-bit atomic memory accesses, this register must be accessible using a 64-bit atomic access.

CNTCV can be accessed through the memory-mapped interface:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Frame	Offset	Instance	Range
Timer	CNTControlBase	0x008	CNTCV	63:0

Accesses to this register are RW.

Component	Frame	Offset	Instance	Range
Timer	CNTReadBase	0x000	CNTCV	63:0

Accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.5.4 CNTEL0ACR, Counter-timer EL0 Access Control Register

The CNTEL0ACR characteristics are:

Purpose

An implementation of CNTEL0ACR in the frame at CNTBaseN controls whether the CNTPCT, CNTVCT, CNTFRQ, EL1 Physical Timer, and Virtual Timer registers are visible in the frame at CNTEL0BaseN.

Configuration

It is IMPLEMENTATION DEFINED whether CNTEL0ACR is implemented in the Core power domain or in the Debug power domain

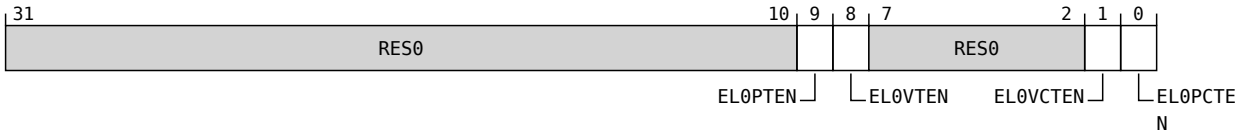
For more information, see 'Power and reset domains for the system level implementation of the Generic Timer'.

Implementation of this register is OPTIONAL.

Attributes

CNTEL0ACR is a 32-bit register.

Field descriptions



Bits [31:10]

Reserved, RES0.

EL0PTEN, bit [9]

Second view read/write access control for the EL1 Physical Timer registers. This bit controls whether the CNTP_CVAL, CNTP_TVAL, and CNTP_CTL registers in the current CNTBaseN frame are also accessible in the corresponding CNTEL0BaseN frame.

EL0PTEN	Meaning
0b0	No access. Registers are RES0 in the second view.
0b1	Access permitted. If the registers are accessible in the current frame, then they are accessible in the second view.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

EL0VTEN, bit [8]

Second view read/write access control for the Virtual Timer registers. This bit controls whether the CNTV_CVAL, CNTV_TVAL, and CNTV_CTL registers in the current CNTBaseN frame are also accessible in the corresponding CNTEL0BaseN frame.

EL0VTEN	Meaning
0b0	No access. Registers are RES0 in the second view.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EL0VTEN	Meaning
0b1	Access permitted. If the registers are accessible in the current frame, then they are accessible in the second view.

The definition of this bit means that, if the Virtual Timer registers are not implemented in the current CNTBaseN frame, then the Virtual Timer register addresses are RES0 in the corresponding CNTEL0BaseN frame, regardless of the value of this bit.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Bits [7:2]

Reserved, RES0.

EL0VCTEN, bit [1]

Second view read access control for CNTVCT and CNTFRQ.

EL0VCTEN	Meaning
0b0	CNTVCT is not visible in the second view. If EL0PCTEN is set to 0, CNTFRQ is not visible in the second view.
0b1	Access permitted. If CNTVCT and CNTFRQ are visible in the current frame, then they are visible in the second view.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

EL0PCTEN, bit [0]

Second view read access control for CNTPCT and CNTFRQ.

EL0PCTEN	Meaning
0b0	CNTPCT is not visible in the second view. If EL0VCTEN is set to 0, CNTFRQ is not visible in the second view.
0b1	Access permitted. If CNTPCT and CNTFRQ are visible in the current frame, then they are visible in the second view.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTEL0ACR

CNTEL0ACR can be implemented in any implemented CNTBaseN frame.

'CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames' describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses. The CNTBaseN frame is always accessible by Secure accesses.

If CNTEL0ACR is not implemented in an implemented CNTBaseN frame:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The register location in that frame is RAZ/WI.
- If the corresponding CNTEL0BaseN frame is implemented, the registers CNTFRQ, CNTP_CTL, CNTP_CVAL, CNTP_TVAL, CNTPCT, CNTV_CTL, CNTV_CVAL, CNTV_TVAL, and CNTVCT are not visible and accesses are RAZ/WI in that frame.

CNTEL0ACR can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTBaseN	0x014	CNTEL0ACR

Accesses to this register are RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.5.5 CNTFID0, Counter Frequency ID

The CNTFID0 characteristics are:

Purpose

Indicates the base frequency of the system counter.

Configuration

It is IMPLEMENTATION DEFINED whether CNTFID0 is implemented in the Core power domain or in the Debug power domain

For more information, see 'Power and reset domains for the system level implementation of the Generic Timer'.

The possible frequencies for the system counter are stored in the Frequency modes table as 32-bit words starting with the base frequency, CNTFID0. For more information, see 'The Frequency modes table'.

The final entry in the Frequency modes table must be followed by a 32-bit word of zero value, to mark the end of the table.

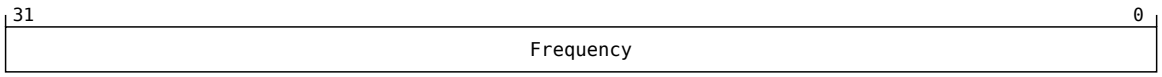
Typically, the Frequency modes table will be in read-only memory. However, a system implementation might use read/write memory for the table, and initialize the table entries as part of its start-up sequence.

If the Frequency modes table is in read/write memory, Arm strongly recommends that the table is not updated once the system is running.

Attributes

CNTFID0 is a 32-bit register.

Field descriptions



Frequency, bits [31:0]

The base frequency of the system counter, in Hz.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTFID0

It is IMPLEMENTATION DEFINED whether this register is RO or RW

In a system that supports Secure and Non-secure physical address spaces, the CNTControlBase frame, which includes this register, is implemented only in the Secure physical address space.

CNTFID0 can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTControlBase	0x020	CNTFID0

Accesses to this register are IMPLEMENTATION DEFINED.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.5.6 CNTFID<n>, Counter Frequency IDs, n > 0, n = 1 - 1003

The CNTFID<n> characteristics are:

Purpose

Indicates alternative system counter frequencies.

Configuration

It is IMPLEMENTATION DEFINED whether CNTFID<n> is implemented in the Core power domain or in the Debug power domain

For more information, see 'Power and reset domains for the system level implementation of the Generic Timer'.

The possible frequencies for the system counter are stored in the Frequency modes table as 32-bit words starting with the base frequency, CNTFID0, see 'The Frequency modes table'.

The number of CNTFID<n> registers is IMPLEMENTATION DEFINED, and the only required CNTFID<n> register is CNTFID0.

The final entry in the Frequency modes table must be followed by a 32-bit word of zero value, to mark the end of the table.

The architecture can support up to 1004 entries in the Frequency modes table, including the zero-word end marker, and the number of entries is IMPLEMENTATION DEFINED up to this limit. For an implementation that includes registers in the IMPLEMENTATION DEFINED register space `0x0C0-0x0FC`, the maximum number of entries in the Frequency modes table is 40, including the zero-word end marker.

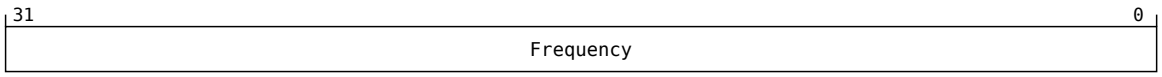
Typically, the Frequency modes table will be in read-only memory. However, a system implementation might use read/write memory for the table, and initialize the table entries as part of its start-up sequence.

If the Frequency modes table is in read/write memory, Arm strongly recommends that the table is not updated once the system is running.

Attributes

CNTFID<n> is a 32-bit register.

Field descriptions



Frequency, bits [31:0]

A system counter frequency, in Hz. Must be an exact divisor of the base frequency. Arm strongly recommends that all frequency values in the Frequency modes table are integer power-of-two divisors of the base frequency.

When the system timer is operating at a lower frequency than the base frequency, the increment applied at each counter update is increased by the ratio of the base frequency and the selected frequency.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTFID<n>

It is IMPLEMENTATION DEFINED whether this register is RO or RW

In a system that supports Secure and Non-secure physical address spaces, the CNTControlBase frame, which includes these registers, is implemented only in the Secure physical address space.

CNTFID<n> can be accessed through the memory-mapped interface:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Frame	Offset	Instance
Timer	CNTControlBase	`0x020 + (4 * n)`	CNTFID<n>

Accesses to this register are IMPLEMENTATION DEFINED.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.5.7 CNTFRQ, Counter-timer Frequency

The CNTFRQ characteristics are:

Purpose

This register is provided so that software can discover the effective frequency of the system counter. The instance of the register in the CNTCTLBase frame must be programmed with this value as part of system initialization. The value of the register is not interpreted by hardware.

Configuration

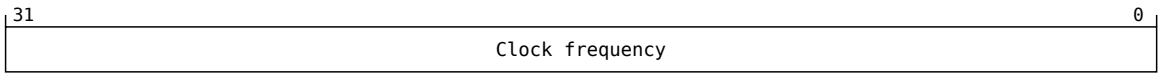
It is IMPLEMENTATION DEFINED whether CNTFRQ is implemented in the Core power domain or in the Debug power domain

For more information see 'Power and reset domains for the system level implementation of the Generic Timer'.

Attributes

CNTFRQ is a 32-bit register.

Field descriptions



ClockFreq, bits [31:0]

Clock frequency. Indicates the effective frequency of the system counter, in Hz.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTFRQ

CNTFRQ must be implemented as an RW register in the CNTCTLBase frame.

In a system that recognizes two Security states, the instance of the register in the CNTCTLBase frame is only accessible by Secure accesses.

CNTFRQ can be implemented as a RO register in any implemented CNTBaseN frame, and in the corresponding CNTEL0BaseN frame.

'CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames' describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses. The CNTBaseN frame is always accessible by Secure accesses.

For an implemented CNTBaseN frame:

- CNTFRQ is accessible in that frame, as a RO register, if the value of CNTACR<n>.RFRQ is 1.
- Otherwise, the CNTFRQ address in that frame is RAZ/WI.

For an implemented CNTEL0BaseN frame:

- CNTFRQ is accessible as a RO register in that frame if both:
  - CNTFRQ is accessible in the corresponding CNTBaseN frame.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

    - Either the value of CNTEL0ACR.EL0VCTEN is 1 or the value of CNTEL0ACR.EL0PCTEN is 1.
- Otherwise, the CNTFRQ address in that frame is RAZ/WI.

CNTFRQ can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTBaseN	0x010	CNTFRQ

Accesses to this register are RO.

Component	Frame	Offset	Instance
Timer	CNTEL0BaseN	0x010	CNTFRQ

Accesses to this register are RO.

Component	Frame	Offset	Instance
Timer	CNTCTLBase	0x000	CNTFRQ

Accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.5.8 CNTID, Counter Identification Register

The CNTID characteristics are:

Purpose

Indicates whether counter scaling is implemented.

Configuration

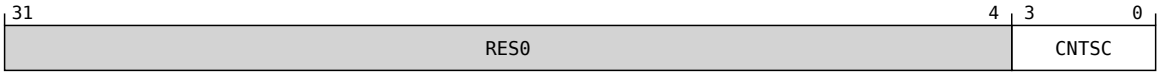
It is IMPLEMENTATION DEFINED whether CNTID is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_CNTSC is implemented. Otherwise, direct accesses to CNTID are RES0.

Attributes

CNTID is a 32-bit register.

Field descriptions



Bits [31:4]

Reserved, RES0.

CNTSC, bits [3:0]

Indicates whether Counter Scaling is implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CNTSC	Meaning
0b0000	Counter scaling is not implemented.
0b0001	Counter scaling is implemented.

All other values are reserved.

Access to this field is RO.

Accessing CNTID

In a system that supports Secure and Non-secure physical address spaces, the CNTControlBase frame, which includes this register, is implemented only in the Secure physical address space.

CNTID can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTControlBase	0x1C	CNTID

Accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.5.9 CNTNSAR, Counter-timer Non-secure Access Register

The CNTNSAR characteristics are:

Purpose

Provides the highest-level control of whether frames CNTBaseN and CNTEL0BaseN are accessible by Non-secure accesses.

Configuration

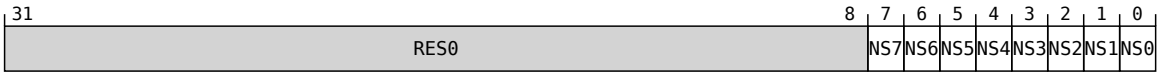
It is IMPLEMENTATION DEFINED whether CNTNSAR is implemented in the Core power domain or in the Debug power domain

For more information, see 'Power and reset domains for the system level implementation of the Generic Timer'.

Attributes

CNTNSAR is a 32-bit register.

Field descriptions



Bits [31:8]

Reserved, RES0.

NS<n> , bits [n], for n = 7 to 0

Non-secure access to frame n.

NS<n>	Meaning
0b0	Secure access only. Behaves as RES0 to Non-secure accesses.
0b1	Secure and Non-secure accesses permitted.

This bit also determines whether, in the CNTCTLBase frame, CNTACR<n> and CNTVOFF<n> are accessible to Non-secure accesses.

If frame CNTBase<n>:

- Is not implemented, then NS<n> is RES0.
- Is not Configurable access, and is accessible only by Secure accesses, then NS<n> is RES0.
- Is not Configurable access, and is accessible by both Secure and Non-secure accesses, then NS<n> is RES1.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTNSAR

In a system that recognizes two Security states, this register is only accessible by Secure accesses.

CNTNSAR can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTCTLBase	0x004	CNTNSAR

Accesses to this register are RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.5.10 CNTP_CTL, Counter-timer Physical Timer Control

The CNTP_CTL characteristics are:

Purpose

Control register for the EL1 physical timer.

Configuration

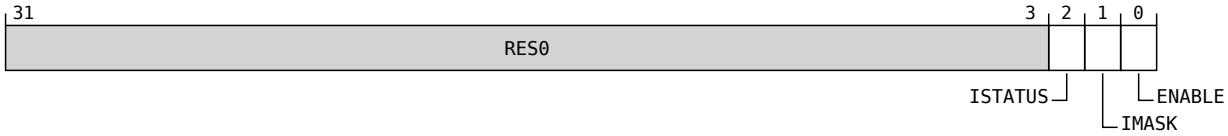
It is IMPLEMENTATION DEFINED whether CNTP_CTL is implemented in the Core power domain or in the Debug power domain

For more information, see 'Power and reset domains for the system level implementation of the Generic Timer'.

Attributes

CNTP_CTL is a 32-bit register.

Field descriptions



Bits [31:3]

Reserved, RES0.

ISTATUS, bit [2]

The status of the timer. This bit indicates whether the timer condition is met:

ISTATUS	Meaning
0b0	Timer condition is not met.
0b1	Timer condition is met.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0, then the timer interrupt is asserted.

When the value of the ENABLE bit is 0, the ISTATUS field is UNKNOWN.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RO.

IMASK, bit [1]

Timer interrupt mask bit. Permitted values are:

IMASK	Meaning
0b0	Timer interrupt is not masked by the IMASK bit.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b1	Timer interrupt is masked by the IMASK bit.

For more information, see the description of the ISTATUS bit.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

ENABLE, bit [0]

Enables the timer. Permitted values are:

ENABLE	Meaning
0b0	Timer disabled.
0b1	Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTP_TVAL continues to count down.

Note

Disabling the output signal might be a power-saving option.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTP_CTL

CNTP_CTL can be implemented in any implemented CNTBaseN frame, and in the corresponding CNTEL0BaseN frame.

'CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames' describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses. The CNTBaseN frame is always accessible by Secure accesses.

For an implemented CNTBaseN frame:

- CNTP_CTL is accessible in that frame if the value of CNTACR<n>.RWPT is 1.
- Otherwise, the CNTP_CTL address in that frame is RAZ/WI.

For an implemented CNTEL0BaseN frame:

- CNTP_CTL is accessible in that frame if both:
  - CNTP_CTL is accessible in the corresponding CNTBaseN frame:
  - The value of CNTEL0ACR.EL0PTEN is 1.
- Otherwise, the CNTP_CTL address in that frame is RAZ/WI.

CNTP_CTL can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTBaseN	0x02C	CNTP_CTL

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accesses to this register are RW.

Component	Frame	Offset	Instance
Timer	CNTEL0BaseN	0x02C	CNTP_CTL

Accesses to this register are RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.5.11 CNTP_CVAL, Counter-timer Physical Timer CompareValue

The CNTP_CVAL characteristics are:

Purpose

Holds the 64-bit compare value for the EL1 physical timer.

Configuration

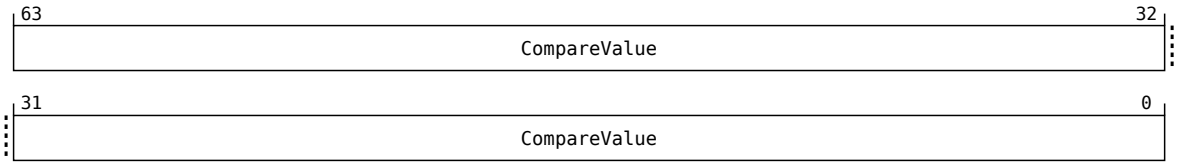
It is IMPLEMENTATION DEFINED whether CNTP_CVAL is implemented in the Core power domain or in the Debug power domain

For more information, see 'Power and reset domains for the system level implementation of the Generic Timer'.

Attributes

CNTP_CVAL is a 64-bit register.

Field descriptions



CompareValue, bits [63:0]

Holds the EL1 physical timer CompareValue.

When CNTP_CTL.ENABLE is 1, the timer condition is met when (CNTPCT - CompareValue) is greater than or equal to zero. This means that CompareValue acts like a 64-bit upcounter timer. When the timer condition is met:

- CNTP_CTL.ISTATUS is set to 1.
- An interrupt is generated if CNTP_CTL.IMASK is 0.

When CNTP_CTL.ENABLE is 0, the timer condition is not met, but CNTPCT continues to count.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTP_CVAL

CNTP_CVAL can be implemented in any implemented CNTBaseN frame, and in the corresponding CNTEL0BaseN frame.

'CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames' describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses. The CNTBaseN frame is always accessible by Secure accesses.

For an implemented CNTBaseN frame:

- CNTP_CVAL is accessible in that frame if the value of CNTACR<n>.RWPT is 1.
- Otherwise, the CNTP_CVAL address in that frame is RAZ/WI.

For an implemented CNTEL0BaseN frame:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- CNTP_CVAL is accessible in that frame if both:
  - CNTP_CVAL is accessible in the corresponding CNTBaseN frame:
  - The value of CNTEL0ACR.EL0PTEN is 1.
- Otherwise, the CNTP_CVAL address in that frame is RAZ/WI.

If the implementation supports 64-bit atomic accesses, then the CNTP_CVAL register must be accessible as an atomic 64-bit value.

CNTP_CVAL can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance	Range
Timer	CNTBaseN	0x020	CNTP_CVAL	31:0

Accesses to this register are RW.

Component	Frame	Offset	Instance	Range
Timer	CNTBaseN	0x024	CNTP_CVAL	63:32

Accesses to this register are RW.

Component	Frame	Offset	Instance	Range
Timer	CNTEL0BaseN	0x020	CNTP_CVAL	31:0

Accesses to this register are RW.

Component	Frame	Offset	Instance	Range
Timer	CNTEL0BaseN	0x024	CNTP_CVAL	63:32

Accesses to this register are RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.5.12 CNTP_TVAL, Counter-timer Physical Timer TimerValue

The CNTP_TVAL characteristics are:

Purpose

Holds the timer value for the EL1 physical timer.

Configuration

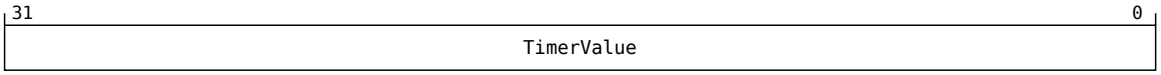
It is IMPLEMENTATION DEFINED whether CNTP_TVAL is implemented in the Core power domain or in the Debug power domain

For more information, see 'Power and reset domains for the system level implementation of the Generic Timer'.

Attributes

CNTP_TVAL is a 32-bit register.

Field descriptions



TimerValue, bits [31:0]

The TimerValue view of the EL1 physical timer.

On a read of this register:

- If CNTP_CTL.ENABLE is 0, the value returned is UNKNOWN.
- If CNTP_CTL.ENABLE is 1, the value returned is (CompareValue - CNTPCT).

On a write of this register, CompareValue is set to (CNTPCT + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When CNTP_CTL.ENABLE is 1, the timer condition is met when (CNTPCT - CompareValue) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:

- CNTP_CTL.ISTATUS is set to 1.
- If CNTP_CTL.IMASK is 0, an interrupt is generated.

When CNTP_CTL.ENABLE is 0, the timer condition is not met, but CNTPCT continues to count, so the TimerValue view appears to continue to count down.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTP_TVAL

CNTP_TVAL can be implemented in any implemented CNTBaseN frame, and in the corresponding CNTEL0BaseN frame.

'CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames' describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses. The CNTBaseN frame is always accessible by Secure accesses.

For an implemented CNTBaseN frame:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- CNTP_TVAL is accessible in that frame if the value of CNTACR<n>.RWPT is 1.
- Otherwise, the CNTP_TVAL address in that frame is RAZ/WI.

For an implemented CNTEL0BaseN frame:

- CNTP_TVAL is accessible in that frame if both:
  - CNTP_TVAL is accessible in the corresponding CNTBaseN frame:
  - The value of CNTEL0ACR.EL0PTEN is 1.
- Otherwise, the CNTP_TVAL address in that frame is RAZ/WI.

CNTP_TVAL can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTBaseN	0x028	CNTP_TVAL

Accesses to this register are RW.

Component	Frame	Offset	Instance
Timer	CNTEL0BaseN	0x028	CNTP_TVAL

Accesses to this register are RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.5.13 CNTPCT, Counter-timer Physical Count

The CNTPCT characteristics are:

Purpose

Holds the 64-bit physical count value.

Configuration

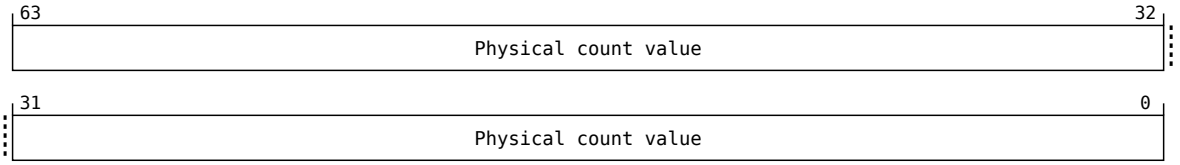
It is IMPLEMENTATION DEFINED whether CNTPCT is implemented in the Core power domain or in the Debug power domain

For more information, see 'Power and reset domains for the system level implementation of the Generic Timer'.

Attributes

CNTPCT is a 64-bit register.

Field descriptions



PhysicalCount, bits [63:0]

Physical count value.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTPCT

CNTPCT can be implemented in any implemented CNTBaseN frame, and in the corresponding CNTEL0BaseN frame, as a RO register.

'CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames' describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses. The CNTBaseN frame is always accessible by Secure accesses.

For an implemented CNTBaseN frame:

- CNTPCT is accessible in that frame, as a RO register, if the value of CNTACR<n>.RPCT is 1.
- Otherwise, the CNTPCT address in that frame is RAZ/WI.

For an implemented CNTEL0BaseN frame:

- CNTPCT is accessible in that frame if both:
  - CNTPCT is accessible in the corresponding CNTBaseN frame.
  - The value of CNTEL0ACR.EL0PCTEN is 1.
- Otherwise, the CNTPCT address in that frame is RAZ/WI.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If the implementation supports 64-bit atomic accesses, then the CNTPCT register must be accessible as an atomic 64-bit value.

CNTPCT can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance	Range
Timer	CNTBaseN	0x000	CNTPCT	31:0

Accesses to this register are RO.

Component	Frame	Offset	Instance	Range
Timer	CNTBaseN	0x004	CNTPCT	63:32

Accesses to this register are RO.

Component	Frame	Offset	Instance	Range
Timer	CNTEL0BaseN	0x000	CNTPCT	31:0

Accesses to this register are RO.

Component	Frame	Offset	Instance	Range
Timer	CNTEL0BaseN	0x004	CNTPCT	63:32

Accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.5.14 CNTSCR, Counter Scale Register

The CNTSCR characteristics are:

Purpose

Enables the counter, controls the counter effective frequency setting, and controls counter behavior during debug.

Configuration

It is IMPLEMENTATION DEFINED whether CNTSCR is implemented in the Core power domain or in the Debug power domain

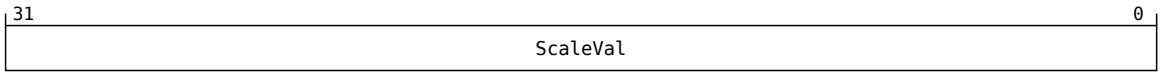
For more information, see 'Power and reset domains for the system level implementation of the Generic Timer'.

This register is present only when FEAT_CNTSC is implemented. Otherwise, direct accesses to CNTSCR are RES0.

Attributes

CNTSCR is a 32-bit register.

Field descriptions



ScaleVal, bits [31:0]

Scale Value

When counter scaling is enabled, ScaleVal is the average amount added to the counter value for one period of the frequency of the Generic counter as described in the CNTFRQ register.

The actual rate of update of the counter value is determined by the counter resolution.

ScaleVal is expressed as an unsigned fixed point number with an 8-bit integer value and a 24-bit fractional value.

CNTSCR.ScaleVal can only be changed when CNTCR.EN == 0. If the value of this field is changed when CNTCR.EN == 1:

- The counter value becomes UNKNOWN.
- The counter value remains UNKNOWN on future ticks of the clock.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTSCR

In a system that supports Secure and Non-secure physical address spaces, the CNTControlBase frame, which includes this register, is implemented only in the Secure physical address space.

CNTSCR can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTControlBase	0x10	CNTSCR

Accesses to this register are RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.5.15 CNTSR, Counter Status Register

The CNTSR characteristics are:

Purpose

Provides counter resolution status information.

Configuration

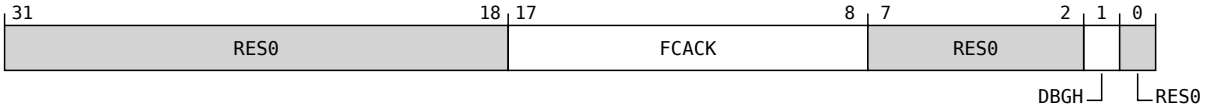
It is IMPLEMENTATION DEFINED whether CNTSR is implemented in the Core power domain or in the Debug power domain

For more information, see 'Power and reset domains for the system level implementation of the Generic Timer'.

Attributes

CNTSR is a 32-bit register.

Field descriptions



Bits [31:18]

Reserved, RES0.

FCACK, bits [17:8]

Frequency Change Acknowledge. Indicates the currently selected entry in the Frequency modes table, see 'The Frequency modes table'.

The reset behavior of this field is:

- On a Timer reset, this field resets to `'0000000000'`.

Bits [7:2]

Reserved, RES0.

DBGH, bit [1]

Indicates whether the counter is halted because the Halt-on-debug signal is asserted:

DBGH	Meaning
`0b0`	Counter is not halted.
`0b1`	Counter is halted.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Bit [0]

Reserved, RES0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing CNTSR

In a system that supports Secure and Non-secure physical address spaces, the CNTControlBase frame, which includes this register, is implemented only in the Secure physical address space.

CNTSR can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTControlBase	`0x004`	CNTSR

Accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.5.16 CNTTIDR, Counter-timer Timer ID Register

The CNTTIDR characteristics are:

Purpose

Indicates the implemented timers in the physical address space, and their features. For each value of N from 0 to 7 it indicates whether:

- Frame CNTBaseN is a view of an implemented timer.
- Frame CNTBaseN has a second view, CNTEL0BaseN.
- Frame CNTBaseN has a virtual timer capability.

Configuration

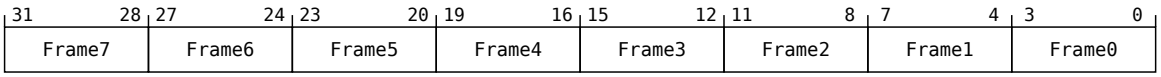
It is IMPLEMENTATION DEFINED whether CNTTIDR is implemented in the Core power domain or in the Debug power domain

For more information, see 'Power and reset domains for the system level implementation of the Generic Timer'.

Attributes

CNTTIDR is a 32-bit register.

Field descriptions



Frame<n> , bits [4n+3:4n], for n = 7 to 0

A 4-bit field indicating the features of frame CNTBase<n>.

Bit[3] of the field is RES0.

Bit[2], the FEL0 subfield, indicates whether frame CNTBase<n> has a second view, CNTEL0Base<n>. The possible values of this bit are:

Bit[2]	Meaning
0b0	Frame<n> does not have a second view. The CNTEL0ACR register in the first view of the frame is RES0.
0b1	Frame<n> has a second view, CNTEL0Base<n>.

If bit[0] is 0, bit[2] is RES0.

Bit[1], the FVI subfield, indicates whether both:

- Frame CNTBase<n> implements the virtual timer registers CNTV_CVAL, CNTV_TVAL, and CNTV_CTL.
- This CNTCTLBase frame implements the virtual timer offset register CNTVOFF<n>.

The possible values of bit[1] are:

Bit[1]	Meaning
0b0	Frame<n> does not have virtual capability. The virtual time and offset registers are RES0.
0b1	Frame<n> has virtual capability. The virtual time and offset registers are implemented.

If bit[0] is 0, bit[1] is RES0.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bit[0], the FI subfield, indicates whether frame CNTBase<n> is implemented. The possible values of this bit are:

Bit[0]	Meaning
`0b0`	Frame<n> is not implemented. All registers associated with the frame are RES0.
`0b1`	Frame<n> is implemented.

Accessing CNTTIDR

In a system that recognizes two Security states, this register is accessible by both Secure and Non-secure accesses.

CNTTIDR can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTCTLBase	`0x008`	CNTTIDR

Accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.5.17 CNTV_CTL, Counter-timer Virtual Timer Control

The CNTV_CTL characteristics are:

Purpose

Control register for the virtual timer.

Configuration

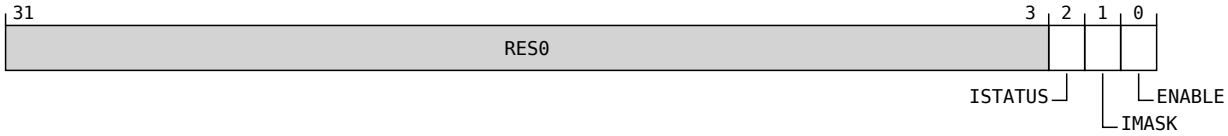
It is IMPLEMENTATION DEFINED whether CNTV_CTL is implemented in the Core power domain or in the Debug power domain

For more information, see 'Power and reset domains for the system level implementation of the Generic Timer'.

Attributes

CNTV_CTL is a 32-bit register.

Field descriptions



Bits [31:3]

Reserved, RES0.

ISTATUS, bit [2]

The status of the timer. This bit indicates whether the timer condition is met:

ISTATUS	Meaning
0b0	Timer condition is not met.
0b1	Timer condition is met.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0, then the timer interrupt is asserted.

When the value of the ENABLE bit is 0, the ISTATUS field is UNKNOWN.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RO.

IMASK, bit [1]

Timer interrupt mask bit. Permitted values are:

IMASK	Meaning
0b0	Timer interrupt is not masked by the IMASK bit.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b1	Timer interrupt is masked by the IMASK bit.

For more information, see the description of the ISTATUS bit.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

ENABLE, bit [0]

Enables the timer. Permitted values are:

ENABLE	Meaning
0b0	Timer disabled.
0b1	Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTV_TVAL continues to count down.

Note

Disabling the output signal might be a power-saving option.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTV_CTL

CNTV_CTL can be implemented in any implemented CNTBaseN frame that has virtual timer capability, and in the corresponding CNTEL0BaseN frame.

'CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames' describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses. The CNTBaseN frame is always accessible by Secure accesses.

For an implemented CNTBaseN frame that has virtual timer capability:

- CNTV_CTL is accessible in that frame if the value of CNTACR<n>.RWVT is 1.
- Otherwise, the CNTV_CTL address in that frame is RAZ/WI.

For an implemented CNTEL0BaseN frame:

- CNTV_CTL is accessible in that frame if both:
  - CNTV_CTL is accessible in the corresponding CNTBaseN frame:
  - The value of CNTEL0ACR.EL0VTEN is 1.
- Otherwise, the CNTV_CTL address in that frame is RAZ/WI.

CNTV_CTL can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTBaseN	0x03C	CNTV_CTL

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accesses to this register are RW.

Component	Frame	Offset	Instance
Timer	CNTEL0BaseN	0x03C	CNTV_CTL

Accesses to this register are RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.5.18 CNTV_CVAL, Counter-timer Virtual Timer CompareValue

The CNTV_CVAL characteristics are:

Purpose

Holds the 64-bit compare value for the virtual timer.

Configuration

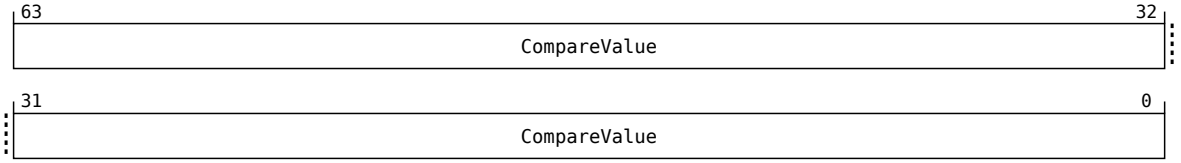
It is IMPLEMENTATION DEFINED whether CNTV_CVAL is implemented in the Core power domain or in the Debug power domain

For more information, see 'Power and reset domains for the system level implementation of the Generic Timer'.

Attributes

CNTV_CVAL is a 64-bit register.

Field descriptions



CompareValue, bits [63:0]

Holds the virtual timer CompareValue.

When CNTV_CTL.ENABLE is 1, the timer condition is met when (CNTVCT - CompareValue) is greater than or equal to zero. This means that CompareValue acts like a 64-bit upcounter timer. When the timer condition is met:

- CNTV_CTL.ISTATUS is set to 1.
- An interrupt is generated if CNTV_CTL.IMASK is 0.

When CNTV_CTL.ENABLE is 0, the timer condition is not met, but CNTVCT continues to count.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTV_CVAL

CNTV_CVAL can be implemented in any implemented CNTBaseN frame that has virtual timer capability, and in the corresponding CNTEL0BaseN frame.

'CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames' describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses. The CNTBaseN frame is always accessible by Secure accesses.

For an implemented CNTBaseN frame that has virtual timer capability:

- CNTV_CVAL is accessible in that frame if the value of CNTACR<n>.RWVT is 1.
- Otherwise, the CNTV_CVAL address in that frame is RAZ/WI.

For an implemented CNTEL0BaseN frame:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- CNTV_CVAL is accessible in that frame if both:
  - CNTV_CVAL is accessible in the corresponding CNTBaseN frame:
  - The value of CNTEL0ACR.EL0VTEN is 1.
- Otherwise, the CNTV_CVAL address in that frame is RAZ/WI.

If the implementation supports 64-bit atomic accesses, then the CNTV_CVAL register must be accessible as an atomic 64-bit value.

CNTV_CVAL can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance	Range
Timer	CNTBaseN	0x030	CNTV_CVAL	31:0

Accesses to this register are RW.

Component	Frame	Offset	Instance	Range
Timer	CNTBaseN	0x034	CNTV_CVAL	63:32

Accesses to this register are RW.

Component	Frame	Offset	Instance	Range
Timer	CNTEL0BaseN	0x030	CNTV_CVAL	31:0

Accesses to this register are RW.

Component	Frame	Offset	Instance	Range
Timer	CNTEL0BaseN	0x034	CNTV_CVAL	63:32

Accesses to this register are RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.5.19 CNTV_TVAL, Counter-timer Virtual Timer TimerValue

The CNTV_TVAL characteristics are:

Purpose

Holds the timer value for the virtual timer.

Configuration

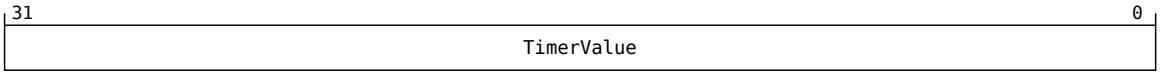
It is IMPLEMENTATION DEFINED whether CNTV_TVAL is implemented in the Core power domain or in the Debug power domain

For more information, see 'Power and reset domains for the system level implementation of the Generic Timer'.

Attributes

CNTV_TVAL is a 32-bit register.

Field descriptions



TimerValue, bits [31:0]

The TimerValue view of the virtual timer.

On a read of this register:

- If CNTV_CTL.ENABLE is 0, the value returned is UNKNOWN.
- If CNTV_CTL.ENABLE is 1, the value returned is (CompareValue - CNTVCT).

On a write of this register, CompareValue is set to (CNTVCT + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When CNTV_CTL.ENABLE is 1, the timer condition is met when (CNTVCT - CompareValue) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:

- CNTV_CTL.ISTATUS is set to 1.
- If CNTV_CTL.IMASK is 0, an interrupt is generated.

When CNTV_CTL.ENABLE is 0, the timer condition is not met, but CNTVCT continues to count, so the TimerValue view appears to continue to count down.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTV_TVAL

CNTV_TVAL can be implemented in any implemented CNTBaseN frame that has virtual timer capability, and in the corresponding CNTEL0BaseN frame.

'CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames' describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses. The CNTBaseN frame is always accessible by Secure accesses.

For an implemented CNTBaseN frame that has virtual timer capability:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- CNTV_TVAL is accessible in that frame if the value of CNTACR<n>.RWVT is 1.
- Otherwise, the CNTV_TVAL address in that frame is RAZ/WI.

For an implemented CNTEL0BaseN frame:

- CNTV_TVAL is accessible in that frame if both:
  - CNTV_TVAL is accessible in the corresponding CNTBaseN frame.
  - The value of CNTEL0ACR.EL0VTEN is 1.
- Otherwise, the CNTV_TVAL address in that frame is RAZ/WI.

CNTV_TVAL can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTBaseN	0x038	CNTV_TVAL

Accesses to this register are RW.

Component	Frame	Offset	Instance
Timer	CNTEL0BaseN	0x038	CNTV_TVAL

Accesses to this register are RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.5.20 CNTVCT, Counter-timer Virtual Count

The CNTVCT characteristics are:

Purpose

Holds the 64-bit virtual count value.

Configuration

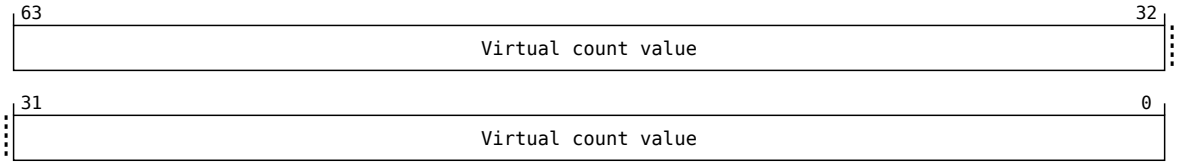
It is IMPLEMENTATION DEFINED whether CNTVCT is implemented in the Core power domain or in the Debug power domain

For more information, see 'Power and reset domains for the system level implementation of the Generic Timer'.

Attributes

CNTVCT is a 64-bit register.

Field descriptions



VirtualCount, bits [63:0]

Virtual count value.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTVCT

CNTVCT can be implemented in any implemented CNTBaseN frame, and in the corresponding CNTEL0BaseN frame, as a RO register.

'CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames' describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses. The CNTBaseN frame is always accessible by Secure accesses.

For an implemented CNTBaseN frame:

- CNTVCT is accessible in that frame, as a RO register, if the value of CNTACR<n>.RVCT is 1.
- Otherwise, the CNTVCT address in that frame is RAZ/WI.

For an implemented CNTEL0BaseN frame:

- CNTVCT is accessible in that frame if both:
  - CNTVCT is accessible in the corresponding CNTBaseN frame.
  - The value of CNTEL0ACR.EL0VCTEN is 1.
- Otherwise, the CNTVCT address in that frame is RAZ/WI.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If the implementation supports 64-bit atomic accesses, then the CNTVCT register must be accessible as an atomic 64-bit value.

CNTVCT can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance	Range
Timer	CNTBaseN	0x008	CNTVCT	31:0

Accesses to this register are RO.

Component	Frame	Offset	Instance	Range
Timer	CNTBaseN	0x00C	CNTVCT	63:32

Accesses to this register are RO.

Component	Frame	Offset	Instance	Range
Timer	CNTEL0BaseN	0x008	CNTVCT	31:0

Accesses to this register are RO.

Component	Frame	Offset	Instance	Range
Timer	CNTEL0BaseN	0x00C	CNTVCT	63:32

Accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.5.21 CNTVOFF, Counter-timer Virtual Offset

The CNTVOFF characteristics are:

Purpose

Holds the 64-bit virtual offset for a CNTBaseN frame that has virtual timer capability. This is the offset between real time and virtual time.

Configuration

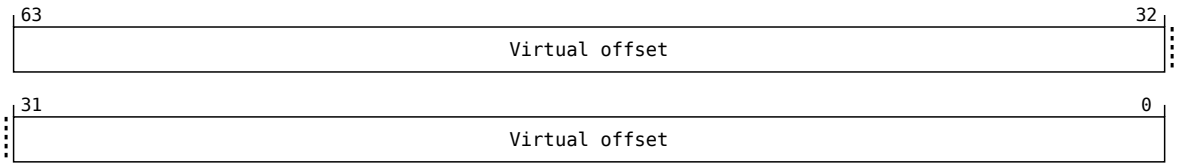
It is IMPLEMENTATION DEFINED whether CNTVOFF is implemented in the Core power domain or in the Debug power domain

For more information, see 'Power and reset domains for the system level implementation of the Generic Timer'.

Attributes

CNTVOFF is a 64-bit register.

Field descriptions



VOffset, bits [63:0]

Virtual offset.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTVOFF

CNTVOFF is implemented, as a RO register, in any implemented CNTBaseN frame that has virtual timer capability.

'CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames' describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses. The CNTBaseN frame is always accessible by Secure accesses.

For an implemented CNTBaseN frame that has virtual timer capability:

- CNTVOFF is accessible in that frame, as a RO register, if the value of CNTACR<n>.RVOFF is 1.
- Otherwise, the CNTVOFF address in that frame is RAZ/WI.

Note

CNTVOFF is never visible in any CNTEL0BaseN frame. This means that the CNTVOFF address in any implemented CNTEL0BaseN frame is RAZ/WI.

In an implementation that supports 64-bit atomic accesses, then the CNTVOFF register must be accessible as an atomic 64-bit value.

CNTVOFF can be accessed through the memory-mapped interface:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Frame	Offset	Range
Timer	CNTBaseN	0x018	31:0

Accesses to this register are RO.

Component	Frame	Offset	Range
Timer	CNTBaseN	0x01C	63:32

Accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.5.22 CNTVOFF<n>, Counter-timer Virtual Offsets, n = 0 - 7

The CNTVOFF<n> characteristics are:

Purpose

Holds the 64-bit virtual offset for frame CNTBase<n>. This is the offset between real time and virtual time.

Configuration

It is IMPLEMENTATION DEFINED whether CNTVOFF<n> is implemented in the Core power domain or in the Debug power domain

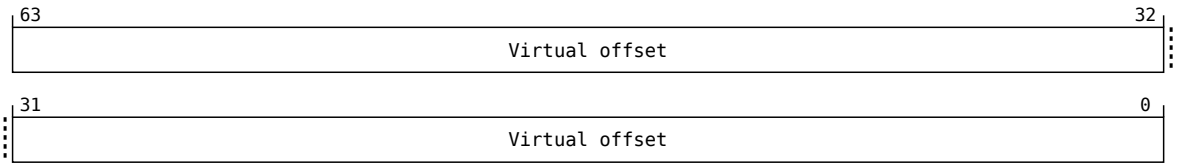
For more information, see 'Power and reset domains for the system level implementation of the Generic Timer'.

Implementation of this register is OPTIONAL.

Attributes

CNTVOFF<n> is a 64-bit register.

Field descriptions



VOffset, bits [63:0]

Virtual offset.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTVOFF<n>

In the CNTCTLBase frame a CNTVOFF<n> register must be implemented, as a RW register, for each CNTBaseN frame that has virtual timer capability. For more information, see 'CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames'.

Note

The value of <n> in an instance of CNTVOFF<n> specifies the value of N for the associated CNTBaseN frame.

In a system that recognizes two Security states, for any CNTVOFF<n> register in the CNTCTLBase frame:

- CNTVOFF<n> is always accessible by Secure accesses.
- CNTNSAR.NS<n> determines whether CNTVOFF<n> is accessible by Non-secure accesses.

The register location of any unimplemented CNTVOFF<n> register in the CNTCTLBase frame is RAZ/WI.

The CNTVOFF<n> register is accessible in the CNTBaseN frame using CNTVOFF.

In an implementation that supports 64-bit atomic accesses, then the CNTVOFF<n> registers must be accessible as atomic 64-bit values.

CNTVOFF<n> can be accessed through the memory-mapped interface:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Frame	Offset	Range
Timer	CNTCTLBase	`0x080 + (8 * n)`	31:0

Accesses to this register are RW.

Component	Frame	Offset	Range
Timer	CNTCTLBase	`0x084 + (8 * n)`	63:32

Accesses to this register are RW.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F4.5.23 CounterID<n>, Counter ID registers, n = 0 - 11

The CounterID<n> characteristics are:

Purpose

IMPLEMENTATION DEFINED identification registers 0 to 11 for the memory-mapped Generic Timer.

Configuration

It is IMPLEMENTATION DEFINED whether CounterID<n> is implemented in the Core power domain or in the Debug power domain

For more information, see 'Power and reset domains for the system level implementation of the Generic Timer'.

These registers are implemented independently in each of the implemented Generic Timer memory-mapped frames.

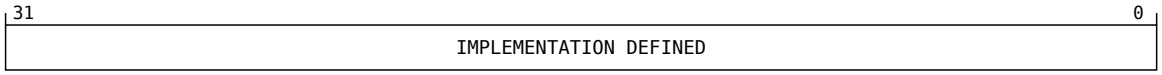
If the implementation of the Counter ID registers requires an architecture version, the value for this version of the Arm Generic Timer is version 0.

The Counter ID registers can be implemented as a set of CoreSight ID registers, comprising Peripheral ID Registers and Component ID Registers. An implementation of these registers for the Generic Timer must use a Component class value of `0xF`.

Attributes

CounterID<n> is a 32-bit register.

Field descriptions



IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

Accessing CounterID<n>

These registers must be implemented, as RO registers, in every implemented Generic Timer memory-mapped frame.

For the CNTCTLBase frame, in a system that recognizes two Security states these registers are accessible by both Secure and Non-secure accesses.

For the CNTControlBase frame, in a system that supports Secure and Non-secure physical address spaces, the frame is implemented only in the Secure physical address space, meaning these registers are implemented only in the Secure physical address space.

For the CNTReadBase frame, these registers are accessible in all physical address spaces.

For the CNTBaseN frames, 'CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames' describes the status fields that identify whether a frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses. The CNTBaseN frame is always accessible by Secure accesses.

CounterID<n> can be accessed through the memory-mapped interface:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Frame	Offset	Instance
Timer	CNTControlBase	`0xFD0 + (4 * n)`	CounterID<n>

Accesses to this register are RO.

Component	Frame	Offset	Instance
Timer	CNTReadBase	`0xFD0 + (4 * n)`	CounterID<n>

Accesses to this register are RO.

Component	Frame	Offset	Instance
Timer	CNTBaseN	`0xFD0 + (4 * n)`	CounterID<n>

Accesses to this register are RO.

Component	Frame	Offset	Instance
Timer	CNTEL0BaseN	`0xFD0 + (4 * n)`	CounterID<n>

Accesses to this register are RO.

Component	Frame	Offset	Instance
Timer	CNTCTLBase	`0xFD0 + (4 * n)`	CounterID<n>

Accesses to this register are RO.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Part G
Architectural Pseudocode

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter G1
R-profile Architecture Pseudocode

This chapter contains pseudocode that describes many features of the A-profile architecture. It contains the following sections:

- Pseudocode for AArch64 operation.
- Shared pseudocode.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.1 Pseudocode for AArch64 operation

This section holds the pseudocode for execution in AArch64 state. Functions that are listed in this section are identified as `AArch64.FunctionName`. Some of these functions have an equivalent AArch32 function, `AArch32.FunctionName`. This section is organized by functional groups, with the functional groups being indicated by hierarchical path names, for example `aarch64/debug/breakpoint`.

The top-level sections of the AArch64 pseudocode hierarchy are:

- aarch64/debug.
- aarch64/exceptions.
- aarch64/functions.
- aarch64/translation.

G1.1.1 aarch64/debug

This section includes the following pseudocode functions:

G1.1.1.1 AArch64.BreakpointMatch

```
// AArch64.BreakpointMatch()
// =========================
// Breakpoint matching in an AArch64 translation regime.
// Returns BreakpointInfo structure that contains a boolean indicating if the match is successful
// and the matched address.

BreakpointInfo AArch64.BreakpointMatch(integer n, bits(64) vaddress, AccessDescriptor accdesc,
                                       integer size)
    assert !ELUsingAArch32(S1TranslationRegime());
    assert n < NumBreakpointsImplemented();
    BreakpointInfo brkptinfo;

    linking_enabled = DBGBCR_EL1[n].BT IN {'0x11', '1xx1'};

    // A breakpoint that has linking enabled does not generate debug events in isolation
    if linking_enabled then
        brkptinfo.match = FALSE;
        return brkptinfo;

    enabled    = IsBreakpointEnabled(n);
    linked     = DBGBCR_EL1[n].BT == '0x01';
    isbreakpnt = TRUE;
    linked_to  = FALSE;
    linked_n = UInt(DBGBCR_EL1[n].LBN);
    state_match = AArch64.StateMatch(DBGBCR_EL1[n].SSC, DBGBCR_EL1[n].HMC,
                                     DBGBCR_EL1[n].PMC, linked, linked_n, isbreakpnt,
                                     vaddress, accdesc);

    value_match = AArch64.BreakpointValueMatch(n, vaddress, linked_to);

    brkptinfo.match = value_match && state_match && enabled;
    return brkptinfo;
```

G1.1.1.2 AArch64.BreakpointValueMatch

```
// AArch64.BreakpointValueMatch()
// ==============================
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// Returns a boolean indicating if the value match is successful.

boolean AArch64.BreakpointValueMatch(integer n_in, bits(64) vaddress, boolean linked_to)

    // "n_in" is the identity of the breakpoint unit to match against.
    // "vaddress" is the current instruction address, ignored if linked_to is TRUE and for Context
    //   matching breakpoints.
    // "linked_to" is TRUE if this is a call from StateMatch for linking.
    integer n = n_in;
    Constraint c;
    bits(5) dbgtype;

    // If a non-existent breakpoint then it is CONSTRAINED UNPREDICTABLE whether this gives
    // no match or the breakpoint is mapped to another UNKNOWN implemented breakpoint.
    if n >= NumBreakpointsImplemented() then
        (c, n) = ConstrainUnpredictableInteger(0, NumBreakpointsImplemented() - 1,
                                               Unpredictable_BPNOTIMPL);
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then return FALSE;

    // If this breakpoint is not enabled, it cannot generate a match.
    // (This could also happen on a call from StateMatch for linking).
    if !IsBreakpointEnabled(n) then return FALSE;

    // If BT is set to a reserved type, behaves either as disabled or as a not-reserved type.
    dbgtype = '0' : DBGBCR_EL1[n].BT;

    (c, dbgtype) = AArch64.ReservedBreakpointType(n, dbgtype);
    if c == Constraint_DISABLED then return FALSE;
    // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

    // Determine what to compare against.
    match_addr      = (dbgtype == 'x0x0x');
    match_vmid      = (dbgtype == 'x10xx');
    match_cid1      = (dbgtype == 'xxx1x');
    match_cid2      = (dbgtype == 'x11xx');
    linking_enabled = (dbgtype IN {'xxx11', 'x1xx1'});

    // If this is a call from StateMatch, return FALSE if the breakpoint is not
    // programmed with linking enabled.
    if linked_to && !linking_enabled then
        return FALSE;

    // If called from BreakpointMatch return FALSE for breakpoint with linking enabled.
    if !linked_to && linking_enabled then
        return FALSE;

    boolean bvr_match  = FALSE;
    boolean bxvr_match = FALSE;
    // Do the comparison.
    if match_addr then
        boolean byte_select_match;
        constant integer byte = UInt(vaddress<1:0>);

        assert byte == 0;                               // "vaddress" is word aligned
        byte_select_match = TRUE;                       // DBGBCR_EL1[n].BAS<byte> is RES1

        // If the DBGBVR_EL1[n].RESS field bits are not a
        // sign extension of the MSB of DBGBVR_EL1[n].VA, it is UNPREDICTABLE whether they
        // appear to be included in the match.
        // If 'vaddress' is outside of the current virtual address space, then the access
        // generates a Translation fault.
        constant AddressSize dbgtop = DebugAddrTop();
        constant boolean unpredictable_ress = (!IsOnes(DBGBVR_EL1[n]<63:dbgtop>) &&
                                               !IsZero(DBGBVR_EL1[n]<63:dbgtop>) &&
                                    ConstrainUnpredictableBool(Unpredictable_DBGxVR_RESS));
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        constant integer cmpmsb = if unpredictable_ress then 63 else dbgtop;
        constant integer cmplsb = 2;
        bvr_match = ((vaddress<cmpmsb:cmplsb> == DBGBVR_EL1[n]<cmpmsb:cmplsb>) &&
                     byte_select_match);

    elsif match_cid1 then
        bvr_match = (PSTATE.EL IN {EL0, EL1} && CONTEXTIDR_EL1<31:0> == DBGBVR_EL1[n]<31:0>);

    if match_vmid then
        constant bits(16) vmid = VMID[];
        bits(16) bvr_vmid;

        if !IsFeatureImplemented(FEAT_VMID16) || VTCR_EL2.VS == '0' then
            bvr_vmid = ZeroExtend(DBGBVR_EL1[n]<39:32>, 16);
        else
            bvr_vmid = DBGBVR_EL1[n]<47:32>;

        bxvr_match = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() && vmid == bvr_vmid);

    elsif match_cid2 then
        bxvr_match = (PSTATE.EL != EL3 && EL2Enabled() &&
                      DBGBVR_EL1[n]<63:32> == CONTEXTIDR_EL2<31:0>);

    bvr_match_valid  = (match_addr || match_cid1);
    bxvr_match_valid = (match_vmid || match_cid2);

    value_match = (!bxvr_match_valid || bxvr_match) && (!bvr_match_valid || bvr_match);

    return value_match;
```

G1.1.1.3 AArch64.ReservedBreakpointType

```
// AArch64.ReservedBreakpointType()
// ================================
// Checks if the given DBGBCR<n>_EL1.BT value is reserved and will
// generate Constrained Unpredictable behavior, otherwise returns Constraint_NONE.

(Constraint, bits(5)) AArch64.ReservedBreakpointType(integer n, bits(5) bt_in)
    bits(5) bt       = bt_in;
    boolean reserved = FALSE;
    context_aware = IsContextAwareBreakpoint(n);

    // Reserved
    if bt == '0010x' then
        reserved = TRUE;

    // Context matching
    if bt != '00x0x' && !context_aware then
        reserved = TRUE;

    // EL2 extension
    if bt == '01xxx' && !HaveEL(EL2) then
        reserved = TRUE;

    // Context matching
    if bt IN {'0011x','011xx'} && !IsFeatureImplemented(FEAT_Debugv8p2) then
        reserved = TRUE;

    if reserved then
        Constraint c;
        (c, bt) = ConstrainUnpredictableBits(Unpredictable_RESBPTYPE, 5);
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            return (c, bits(5) UNKNOWN);
        // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

    return (Constraint_NONE, bt);
```

G1.1.1.4 AArch64.StateMatch

```
// AArch64.StateMatch()
// ====================
// Determine whether a breakpoint or watchpoint is enabled in the current mode and state.

boolean AArch64.StateMatch(bits(2) ssc_in, bit hmc_in,
                           bits(2) pxc_in, boolean linked_in, integer linked_n_in,
                           boolean isbreakpnt, bits(64) vaddress, AccessDescriptor accdesc)
    // "ssc_in","hmc_in","pxc_in" are the control fields from
    // the DBGBCR_EL1[n] or DBGWCR_EL1[n] register.
    // "linked_in" is TRUE if this is a linked breakpoint/watchpoint type.
    // "linked_n_in" is the linked breakpoint number from the DBGBCR_EL1[n] or
    // DBGWCR_EL1[n] register.
    // "isbreakpnt" is TRUE for breakpoints, FALSE for watchpoints.
    // "accdesc" describes the properties of the access being matched.
    bits(2) ssc      = ssc_in;
    bit hmc          = hmc_in;
    bits(2) pxc      = pxc_in;
    boolean linked   = linked_in;
    integer linked_n = linked_n_in;

    // If parameters are set to a reserved type, behaves as either disabled or a defined type
    Constraint c;
    (c, ssc, hmc, pxc) = CheckValidStateMatch(ssc, hmc, pxc, isbreakpnt);
    if c == Constraint_DISABLED then return FALSE;
    // Otherwise the hmc,ssc,ssce,pxc values are either valid or the values returned by
    // CheckValidStateMatch are valid.

    EL3_match = HaveEL(EL3) && hmc == '1' && ssc<0> == '0';
    EL2_match = HaveEL(EL2) && ((hmc == '1' && (ssc:pxc != '1000')) || ssc == '11');
    EL1_match = pxc<0> == '1';
    EL0_match = pxc<1> == '1';

    boolean priv_match;
    case accdesc.el of
        when EL2  priv_match = EL2_match;
        when EL1  priv_match = EL1_match;
        when EL0  priv_match = EL0_match;

    // Security state match
    boolean ss_match;
    case ssc of
        when '00' ss_match = TRUE;                                      // Both
        when '01' ss_match = accdesc.ss == SS_NonSecure;                // Non-secure only
        when '10' ss_match = accdesc.ss == SS_Secure;                   // Secure only
        when '11' ss_match = (hmc == '1' || accdesc.ss == SS_Secure);   // HMC=1 -> Both,
                                                                        // HMC=0 -> Secure only

    boolean linked_match = FALSE;

    if linked then
        // "linked_n" must be an enabled context-aware breakpoint unit. If it is not context-aware
        // then it is CONSTRAINED UNPREDICTABLE whether this gives no match, gives a match without
        // linking, or linked_n is mapped to some UNKNOWN breakpoint that is context-aware.
        if !IsContextAwareBreakpoint(linked_n) then
            (first_ctx_cmp, last_ctx_cmp) = ContextAwareBreakpointRange();
            (c, linked_n) = ConstrainUnpredictableInteger(first_ctx_cmp, last_ctx_cmp,
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                                                    Unpredictable_BPNOTCTXCMP);
            assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};

            case c of
                when Constraint_DISABLED  return FALSE;      // Disabled
                when Constraint_NONE      linked = FALSE;    // No linking
                // Otherwise ConstrainUnpredictableInteger returned a context-aware breakpoint

    if linked then
        linked_to = TRUE;
        linked_match = AArch64.BreakpointValueMatch(linked_n, bits(64) UNKNOWN, linked_to);

    return priv_match && ss_match && (!linked || linked_match);
```

G1.1.1.5 DebugAddrTop

```
// DebugAddrTop()
// ==============
// Returns the value for the top bit used in Breakpoint and Watchpoint address comparisons.

AddressSize DebugAddrTop()
    if IsFeatureImplemented(FEAT_LVA) then
        return 52;
    else
        return 48;
```

G1.1.1.6 IsBreakpointEnabled

```
// IsBreakpointEnabled()
// =====================
// Returns TRUE if the effective value of DBGBCR_EL1[n].E is '1', and FALSE otherwise.

boolean IsBreakpointEnabled(integer n)
    return DBGBCR_EL1[n].E == '1';
```

G1.1.1.7 AArch64.GenerateDebugExceptions

```
// AArch64.GenerateDebugExceptions()
// =================================

boolean AArch64.GenerateDebugExceptions()
    ss = CurrentSecurityState();
    return AArch64.GenerateDebugExceptionsFrom(PSTATE.EL, ss, PSTATE.D);
```

G1.1.1.8 AArch64.GenerateDebugExceptionsFrom

```
// AArch64.GenerateDebugExceptionsFrom()
// =====================================

boolean AArch64.GenerateDebugExceptionsFrom(bits(2) from_el, SecurityState from_state, bit mask)

    if OSLSR_EL1.OSLK == '1' || DoubleLockStatus() || Halted() then
        return FALSE;

    route_to_el2 = (HaveEL(EL2) && (from_state != SS_Secure || IsSecureEL2Enabled()) &&
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                    (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));
    target = (if route_to_el2 then EL2 else EL1);
    enabled = TRUE;

    if from_el == target then
        enabled = enabled && MDSCR_EL1.KDE == '1' && mask == '0';
    else
        enabled = enabled && UInt(target) > UInt(from_el);

    return enabled;
```

G1.1.1.9 AArch64.IncrementCycleCounter

```
// AArch64.IncrementCycleCounter()
// ===============================
// Increment the cycle counter and possibly set overflow bits.

AArch64.IncrementCycleCounter()
    if !CountPMUEvents(CYCLE_COUNTER_ID) then return;

    constant integer old_value = UInt(PMCCNTR_EL0);
    constant integer new_value = old_value + 1;
    PMCCNTR_EL0 = new_value<63:0>;

    if old_value<64> != new_value<64> then
        PMOVSSET_EL0.C = '1';

    return;
```

G1.1.1.10 AArch64.IncrementEventCounter

```
// AArch64.IncrementEventCounter()
// ===============================
// Increment the specified event counter 'idx' by the specified amount 'increment'.

AArch64.IncrementEventCounter(integer idx, integer increment)
    integer old_value;
    integer new_value;

    old_value = UInt(PMEVCNTR_EL0[idx]);
    new_value = old_value + increment;

    PMEVCNTR_EL0[idx] = ZeroExtend(new_value<31:0>, 64);
    constant integer ovflw = 32;
    if old_value<64:ovflw> != new_value<64:ovflw> then
        PMOVSSET_EL0<idx> = '1';
        // Check for the CHAIN event from an even counter
        if (idx<0> == '0' && idx + 1 < NUM_PMU_COUNTERS &&
              (GetPMUCounterRange(idx) == GetPMUCounterRange(idx+1) ||
                 ConstrainUnpredictableBool(Unpredictable_COUNT_CHAIN))) then
            // If PMU counters idx and idx+1 are not in same range,
            // it is CONSTRAINED UNPREDICTABLE if CHAIN event is counted
            PMUEvent(PMU_EVENT_CHAIN, 1, idx + 1);

    return;
```

G1.1.1.11 AArch64.PMUCycle

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// AArch64.PMUCycle()
// ==================
// Called at the end of each cycle to increment event counters and
// check for PMU overflow. In pseudocode, a cycle ends after the
// execution of the operational pseudocode.

AArch64.PMUCycle()
    if !IsFeatureImplemented(FEAT_PMUv3) then
        return;
    PMUEvent(PMU_EVENT_CPU_CYCLES);
    constant integer counters = NUM_PMU_COUNTERS;
    if counters != 0 then
        for idx = 0 to counters - 1
            if CountPMUEvents(idx) then
                constant integer accumulated = PMUEventAccumulator[idx];
                AArch64.IncrementEventCounter(idx, accumulated);
            PMUEventAccumulator[idx] = 0;
    AArch64.IncrementCycleCounter();
    CheckForPMUOverflow();
```

G1.1.1.12 TimeStamp

```
// TimeStamp
// =========

enumeration TimeStamp {
    TimeStamp_None,              // No timestamp
    TimeStamp_CoreSight,         // CoreSight time (IMPLEMENTATION DEFINED)
    TimeStamp_Physical,          // Physical counter value with no offset
    TimeStamp_Virtual  };       // Physical counter value minus CNTVOFF_EL2
```

G1.1.1.13 AArch64.TakeExceptionInDebugState

```
// AArch64.TakeExceptionInDebugState()
// ===================================
// Take an exception in Debug state to an Exception level using AArch64.

AArch64.TakeExceptionInDebugState(bits(2) target_el, ExceptionRecord exception_in)
    assert HaveEL(target_el) && !ELUsingAArch32(target_el) && UInt(target_el) >= UInt(PSTATE.EL);
    assert target_el != EL3 || EDSCR.SDD == '0';
    ExceptionRecord except = exception_in;
    boolean sync_errors;
    if IsFeatureImplemented(FEAT_IESB) then
        sync_errors = SCTLR_EL[target_el].IESB == '1';
        // The Effective value of SCTLR[].IESB might be zero in Debug state.
        if !ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
            sync_errors = FALSE;
    else
        sync_errors = FALSE;

    // Synchronize the context, including Instruction Fetch Barrier effect
    SynchronizeContext();

    // If coming from AArch32 state, the top parts of the X[] registers might be set to zero
    from_32 = UsingAArch32();
    if from_32 then AArch64.MaybeZeroRegisterUppers();

    AArch64.ReportException(except, target_el);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    PSTATE.EL = target_el;
    PSTATE.nRW = '0';
    PSTATE.SP = '1';

    SPSR_ELx[] = bits(64) UNKNOWN;
    ELR_ELx[] = bits(64) UNKNOWN;
    // PSTATE.{SS,D,A,I,F} are not observable and ignored in Debug state, so behave as if UNKNOWN.
    PSTATE.<SS,D,A,I,F> = bits(5) UNKNOWN;
    if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = bit UNKNOWN;
    PSTATE.IL = '0';
    if IsFeatureImplemented(FEAT_UAO) then PSTATE.UAO = '0';
    if IsFeatureImplemented(FEAT_PAN) && PSTATE.EL == EL1 && SCTLR_EL1.SPAN == '0' then
        PSTATE.PAN = '1';
    DLR_EL0 = bits(64) UNKNOWN;
    DSPSR_EL0 = bits(64) UNKNOWN;

    EDSCR.ERR = '1';
    UpdateEDSCRFields();                        // Update EDSCR PE state flags.

    if sync_errors then
        SynchronizeErrors();

    EndOfInstruction();
```

G1.1.1.14 AArch64.WatchpointByteMatch

```
// AArch64.WatchpointByteMatch()
// =============================

boolean AArch64.WatchpointByteMatch(integer n, bits(64) vaddress)
    constant AddressSize dbgtop = DebugAddrTop();
    constant integer cmpbottom = if DBGWVR_EL1[n]<2> == '1' then 2 else 3; // Word or doubleword
    bottom = cmpbottom;
    constant integer select = UInt(vaddress<cmpbottom-1:0>);
    byte_select_match = (DBGWCR_EL1[n].BAS<select> != '0');
    mask = UInt(DBGWCR_EL1[n].MASK);

    // If DBGWCR_EL1[n].MASK is a nonzero value and DBGWCR_EL1[n].BAS is not set to '11111111', or
    // DBGWCR_EL1[n].BAS specifies a non-contiguous set of bytes behavior is CONSTRAINED
    // UNPREDICTABLE.
    if mask > 0 && !IsOnes(DBGWCR_EL1[n].BAS) then
        byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPMASKANDBAS);
    else
        LSB = (DBGWCR_EL1[n].BAS AND NOT(DBGWCR_EL1[n].BAS - 1));  MSB = (DBGWCR_EL1[n].BAS + LSB);
        if !IsZero(MSB AND (MSB - 1)) then          // Not contiguous
            byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPBASCONTIGUOUS);
            bottom = 3;                             // For the whole doubleword

    // If the address mask is set to a reserved value, the behavior is CONSTRAINED UNPREDICTABLE.
    if mask > 0 && mask <= 2 then
        Constraint c;
        (c, mask) = ConstrainUnpredictableInteger(3, 31, Unpredictable_RESWPMASK);
        assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
        case c of
            when Constraint_DISABLED  return FALSE; // Disabled
            when Constraint_NONE      mask = 0;     // No masking
            // Otherwise the value returned by ConstrainUnpredictableInteger is a not-reserved value

    // If the DBGWVR_EL1[n].RESS field bits are not a
    // sign extension of the MSB of DBGWVR_EL1[n].VA, it is UNPREDICTABLE whether they
    // appear to be included in the match.
    constant boolean unpredictable_ress = (!IsOnes(DBGWVR_EL1[n]<63:dbgtop>) &&
                                           !IsZero(DBGWVR_EL1[n]<63:dbgtop>) &&
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                                  ConstrainUnpredictableBool(Unpredictable_DBGxVR_RESS));
    constant integer cmpmsb = if unpredictable_ress then 63 else dbgtop;
    constant integer cmplsb = if mask > bottom then mask else bottom;
    constant integer bottombit = bottom;
    boolean WVR_match = (vaddress<cmpmsb:cmplsb> == DBGWVR_EL1[n]<cmpmsb:cmplsb>);
    if mask > bottom then
        // If masked bits of DBGWVR_EL1[n] are not zero, the behavior is CONSTRAINED UNPREDICTABLE.
        if WVR_match && !IsZero(DBGWVR_EL1[n]<cmplsb-1:bottombit>) then
            WVR_match = ConstrainUnpredictableBool(Unpredictable_WPMASKEDBITS);

    return (WVR_match && byte_select_match);
```

G1.1.1.15 AArch64.WatchpointMatch

```
// AArch64.WatchpointMatch()
// =========================
// Watchpoint matching in an AArch64 translation regime.

WatchpointInfo AArch64.WatchpointMatch(integer n, bits(64) vaddress, integer size,
                                       AccessDescriptor accdesc)
    assert !ELUsingAArch32(S1TranslationRegime());
    assert n < NumWatchpointsImplemented();

    constant boolean enabled       = IsWatchpointEnabled(n);
    linked = DBGWCR_EL1[n].WT == '1';
    isbreakpnt = FALSE;
    linked_n = UInt(DBGWCR_EL1[n].LBN);
    state_match = AArch64.StateMatch(DBGWCR_EL1[n].SSC, DBGWCR_EL1[n].HMC, DBGWCR_EL1[n].PAC,
                                     linked, linked_n, isbreakpnt, PC64, accdesc);

    WatchpointInfo watchptinfo;
    boolean ls_match;
    case DBGWCR_EL1[n].LSC<1:0> of
        when '00' ls_match = FALSE;
        when '01' ls_match = accdesc.read;
        when '10' ls_match = accdesc.write || accdesc.acctype == AccessType_DC;
        when '11' ls_match = TRUE;

    boolean value_match = FALSE;
    watchptinfo.vaddress = vaddress;
    for byte = 0 to size - 1
        if (!value_match &&
              !AddressInNaturallyAlignedBlock(watchptinfo.vaddress, vaddress + byte)) then
            // Watchpoint should report an address which is in
            // the naturally aligned block of the matched address.
            watchptinfo.vaddress = vaddress + byte;
        value_match = value_match || AArch64.WatchpointByteMatch(n, vaddress + byte);

    watchptinfo.watchpt_num = n;
    watchptinfo.value_match = value_match && state_match && ls_match && enabled;
    return watchptinfo;
```

G1.1.1.16 IsWatchpointEnabled

```
// IsWatchpointEnabled()
// =====================
// Returns TRUE if the effective value of DBGWCR_EL1[n].E is '1', and FALSE otherwise.

boolean IsWatchpointEnabled(integer n)
    return DBGWCR_EL1[n].E == '1';
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.1.2 aarch64/exceptions

This section includes the following pseudocode functions:

G1.1.2.1 AArch64.Abort

```
// AArch64.Abort()
// ===============
// Abort and Debug exception handling in an AArch64 translation regime.

AArch64.Abort(FaultRecord fault)
    if IsDebugException(fault) then
        if fault.accessdesc.acctype == AccessType_IFETCH then
            AArch64.BreakpointException(fault);
        else
            AArch64.WatchpointException(fault);
    elsif IsExternalAbort(fault) && !IsExternalSyncAbort(fault) then
        PendSErrorInterrupt(fault);
    elsif fault.accessdesc.acctype == AccessType_IFETCH then
        AArch64.InstructionAbort(fault);
    else
        AArch64.DataAbort(fault);
```

G1.1.2.2 AArch64.AbortSyndrome

```
// AArch64.AbortSyndrome()
// =======================
// Creates an exception syndrome record for Abort and Watchpoint exceptions
// from an AArch64 translation regime.

ExceptionRecord AArch64.AbortSyndrome(Exception exceptype, FaultRecord fault, bits(2) target_el)
    except = ExceptionSyndrome(exceptype);

    except.syndrome = AArch64.FaultSyndrome(exceptype, fault, target_el);
    if exceptype == Exception_Watchpoint then
        except.vaddress = fault.watchptinfo.vaddress;
    else
        except.vaddress = ZeroExtend(fault.vaddress, 64);

    if IPAValid(fault) then
        except.ipavalid = TRUE;
        except.NS = if fault.ipaddress.paspace == PAS_NonSecure then '1' else '0';
        except.ipaddress = fault.ipaddress.address;
    else
        except.ipavalid = FALSE;

    return except;
```

G1.1.2.3 AArch64.CheckPCAlignment

```
// AArch64.CheckPCAlignment()
// ==========================

AArch64.CheckPCAlignment()
    constant bits(64) pc = ThisInstrAddr(64);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if pc<1:0> != '00' then
        AArch64.PCAlignmentFault();
```

G1.1.2.4 AArch64.DataAbort

```
// AArch64.DataAbort()
// ===================

AArch64.DataAbort(FaultRecord fault)
    bits(2) target_el;
    if IsExternalAbort(fault) then
        target_el = SyncExternalAbortTarget(fault);
    else
        route_to_el2 = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} &&
                        (HCR_EL2.TGE == '1' ||
                         IsSecondStage(fault)));

        if PSTATE.EL == EL2 || route_to_el2 then
            target_el = EL2;
        else
            target_el = EL1;

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    constant integer vect_offset = 0x0;

    except = AArch64.AbortSyndrome(Exception_DataAbort, fault, target_el);
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

G1.1.2.5 AArch64.InstructionAbort

```
// AArch64.InstructionAbort()
// ==========================

AArch64.InstructionAbort(FaultRecord fault)
    bits(2) target_el;
    if IsExternalAbort(fault) then
        target_el = SyncExternalAbortTarget(fault);
    else
        route_to_el2 = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} &&
                        (HCR_EL2.TGE == '1' ||
                         IsSecondStage(fault)));

        if PSTATE.EL == EL2 || route_to_el2 then
            target_el = EL2;
        else
            target_el = EL1;

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    integer vect_offset;

    vect_offset = 0x0;

    constant ExceptionRecord except = AArch64.AbortSyndrome(Exception_InstructionAbort, fault,
                                                            target_el);
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

G1.1.2.6 AArch64.PCAlignmentFault

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// AArch64.PCAlignmentFault()
// ==========================
// Called on unaligned program counter in AArch64 state.

AArch64.PCAlignmentFault()

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception_PCAlignment);
    except.vaddress = ThisInstrAddr(64);
    bits(2) target_el = EL1;
    if UInt(PSTATE.EL) > UInt(EL1) then
        target_el = PSTATE.EL;
    elsif EL2Enabled() && HCR_EL2.TGE == '1' then
        target_el = EL2;
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

G1.1.2.7 AArch64.SPAlignmentFault

```
// AArch64.SPAlignmentFault()
// ==========================
// Called on an unaligned stack pointer in AArch64 state.

AArch64.SPAlignmentFault()

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception_SPAlignment);

    bits(2) target_el = EL1;
    if UInt(PSTATE.EL) > UInt(EL1) then
        target_el = PSTATE.EL;
    elsif EL2Enabled() && HCR_EL2.TGE == '1' then
        target_el = EL2;
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

G1.1.2.8 AArch64.TakePhysicalFIQException

```
// AArch64.TakePhysicalFIQException()
// ==================================

AArch64.TakePhysicalFIQException()

    route_to_el3 = FALSE;
    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || HCR_EL2.FMO == '1'));
    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x100;
    except = ExceptionSyndrome(Exception_FIQ);

    if route_to_el3 then
        AArch64.TakeException(EL3, except, preferred_exception_return, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_el2 then
        assert PSTATE.EL != EL3;
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        assert PSTATE.EL IN {EL0, EL1};
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```

G1.1.2.9 AArch64.TakePhysicalIRQException

```
// AArch64.TakePhysicalIRQException()
// ==================================
// Take an enabled physical IRQ exception.

AArch64.TakePhysicalIRQException()

    route_to_el3 = FALSE;
    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || HCR_EL2.IMO == '1'));
    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x80;

    except = ExceptionSyndrome(Exception_IRQ);

    if route_to_el3 then
        AArch64.TakeException(EL3, except, preferred_exception_return, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_el2 then
        assert PSTATE.EL != EL3;
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        assert PSTATE.EL IN {EL0, EL1};
        AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```

G1.1.2.10 AArch64.TakePhysicalSErrorException

```
// AArch64.TakePhysicalSErrorException()
// =====================================

AArch64.TakePhysicalSErrorException(boolean implicit_esb)
    boolean masked;
    bits(2) target_el;

    (masked, target_el) = PhysicalSErrorTarget();
    assert !masked;

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    constant integer vect_offset = 0x180;
    constant FaultRecord fault = GetPendingPhysicalSError();
    except = ExceptionSyndrome(Exception_SError);
    constant boolean is_esb = FALSE;
    constant bits(25) syndrome = AArch64.PhysicalSErrorSyndrome(is_esb, implicit_esb);

    if IsSErrorEdgeTriggered() then
        ClearPendingPhysicalSError();

    except.syndrome.iss = syndrome;
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

G1.1.2.11 AArch64.TakeVirtualFIQException

```
// AArch64.TakeVirtualFIQException()
// =================================

AArch64.TakeVirtualFIQException()
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    assert HCR_EL2.TGE == '0' && HCR_EL2.FMO == '1';  // Virtual IRQ enabled if TGE==0 and FMO==1

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x100;

    except = ExceptionSyndrome(Exception_FIQ);

    AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```

G1.1.2.12 AArch64.TakeVirtualIRQException

```
// AArch64.TakeVirtualIRQException()
// =================================

AArch64.TakeVirtualIRQException()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    assert HCR_EL2.TGE == '0' && HCR_EL2.IMO == '1';  // Virtual IRQ enabled if TGE==0 and IMO==1

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x80;

    except = ExceptionSyndrome(Exception_IRQ);

    AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```

G1.1.2.13 AArch64.TakeVirtualSErrorException

```
// AArch64.TakeVirtualSErrorException()
// ====================================

AArch64.TakeVirtualSErrorException()

    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    assert HCR_EL2.TGE == '0' && HCR_EL2.AMO == '1';  // Virtual SError enabled if TGE==0 and AMO==1

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x180;
    except = ExceptionSyndrome(Exception_SError);

    if IsFeatureImplemented(FEAT_RAS) then
        except.syndrome.iss<24>   = VSESR_EL2.IDS;
        except.syndrome.iss<23:0> = VSESR_EL2.ISS;
    else
        constant bits(25) syndrome = bits(25) IMPLEMENTATION_DEFINED "Virtual SError syndrome";
        impdef_syndrome = syndrome<24> == '1';
        if impdef_syndrome then except.syndrome.iss = syndrome;

    ClearPendingVirtualSError();
    AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```

G1.1.2.14 AArch64.BreakpointException

```
// AArch64.BreakpointException()
// =============================

AArch64.BreakpointException(FaultRecord fault)
    assert PSTATE.EL != EL3;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    bits(2) target_el;
    vect_offset = 0x0;
    target_el = if (PSTATE.EL == EL2 || route_to_el2) then EL2 else EL1;

    vaddress = bits(64) UNKNOWN;
    except = AArch64.AbortSyndrome(Exception_Breakpoint, fault, target_el);
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

G1.1.2.15 AArch64.SoftwareBreakpoint

```
// AArch64.SoftwareBreakpoint()
// ============================

AArch64.SoftwareBreakpoint(bits(16) immediate)

    route_to_el2 = (PSTATE.EL IN {EL0, EL1} &&
                    EL2Enabled() && (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception_SoftwareBreakpoint);
    except.syndrome.iss<15:0> = immediate;

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, except, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```

G1.1.2.16 AArch64.SoftwareStepException

```
// AArch64.SoftwareStepException()
// ===============================

AArch64.SoftwareStepException()
    assert PSTATE.EL != EL3;

    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception_SoftwareStep);
    if SoftwareStep_DidNotStep() then
        except.syndrome.iss<24> = '0';
    else
        except.syndrome.iss<24> = '1';
        except.syndrome.iss<6> = if SoftwareStep_SteppedEX() then '1' else '0';
    except.syndrome.iss<5:0> = '100010';                // IFSC = Debug Exception

    if PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```

G1.1.2.17 AArch64.VectorCatchException

```
// AArch64.VectorCatchException()
// ==============================
// Vector Catch taken from EL0 or EL1 to EL2. This can only be called when debug exceptions are
// being routed to EL2, as Vector Catch is a legacy debug event.

AArch64.VectorCatchException(FaultRecord fault)
    assert PSTATE.EL != EL2;
    assert EL2Enabled() && (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1');

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    vaddress = bits(64) UNKNOWN;
    except = AArch64.AbortSyndrome(Exception_VectorCatch, fault, EL2);

    AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
```

G1.1.2.18 AArch64.WatchpointException

```
// AArch64.WatchpointException()
// =============================

AArch64.WatchpointException(FaultRecord fault)
    assert PSTATE.EL != EL3;

    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    bits(2) target_el;
    vect_offset = 0x0;
    target_el = if (PSTATE.EL == EL2 || route_to_el2) then EL2 else EL1;

    except = AArch64.AbortSyndrome(Exception_Watchpoint, fault, target_el);
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

G1.1.2.19 AArch64.ExceptionClass

```
// AArch64.ExceptionClass()
// ========================
// Returns the Exception Class and Instruction Length fields to be reported in ESR

(integer,bit) AArch64.ExceptionClass(Exception exceptype, bits(2) target_el)

    il_is_valid = TRUE;
    from_32 = UsingAArch32();
    integer ec;
    case exceptype of
        when Exception_Uncategorized        ec = 0x00; il_is_valid = FALSE;
        when Exception_WFxTrap              ec = 0x01;
        when Exception_CP15RTTrap           ec = 0x03; assert from_32;
        when Exception_CP15RRTTrap          ec = 0x04; assert from_32;
        when Exception_CP14RTTrap           ec = 0x05; assert from_32;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when Exception_CP14DTTrap            ec = 0x06; assert from_32;
        when Exception_AdvSIMDFPAccessTrap   ec = 0x07;
        when Exception_FPIDTrap              ec = 0x08;
        when Exception_PACTrap               ec = 0x09;
        when Exception_CP14RRTTrap           ec = 0x0C; assert from_32;
        when Exception_IllegalState          ec = 0x0E; il_is_valid = FALSE;
        when Exception_SupervisorCall        ec = if from_32 then 0x11 else 0x15;
        when Exception_HypervisorCall        ec = if from_32 then 0x12 else 0x16;
        when Exception_MonitorCall           ec = if from_32 then 0x13 else 0x17;
        when Exception_SystemRegisterTrap    ec = 0x18; assert !from_32;
        when Exception_PACFail               ec = 0x1C; assert !from_32;
        when Exception_InstructionAbort
            ec = if target_el == PSTATE.EL then 0x21 else 0x20;
            il_is_valid = FALSE;
        when Exception_PCAlignment           ec = 0x22; il_is_valid = FALSE;
        when Exception_DataAbort             ec = if target_el == PSTATE.EL then 0x25 else 0x24;
        when Exception_SPAlignment           ec = 0x26; il_is_valid = FALSE; assert !from_32;
        when Exception_FPTrappedException    ec = if from_32 then 0x28 else 0x2C;
        when Exception_SError                ec = 0x2F; il_is_valid = FALSE;
        when Exception_Breakpoint
            ec = if target_el == PSTATE.EL then 0x31 else 0x30;
            il_is_valid = FALSE;
        when Exception_SoftwareStep
            ec = if target_el == PSTATE.EL then 0x33 else 0x32;
            il_is_valid = FALSE;
        when Exception_Watchpoint
            ec = if target_el == PSTATE.EL then 0x35 else 0x34;
            il_is_valid = FALSE;
        when Exception_SoftwareBreakpoint    ec = if from_32 then 0x38 else 0x3C;
        when Exception_VectorCatch           ec = 0x3A; il_is_valid = FALSE; assert from_32;
        otherwise                            Unreachable();

    bit il;
    if il_is_valid then
        il = if ThisInstrLength() == 32 then '1' else '0';
    else
        il = '1';
    assert from_32 || il == '1';            // AArch64 instructions always 32-bit

    return (ec,il);
```

G1.1.2.20 AArch64.ReportException

```
// AArch64.ReportException()
// =========================
// Report syndrome information for exception taken to AArch64 state.

AArch64.ReportException(ExceptionRecord except, bits(2) target_el)

    constant Exception exceptype = except.exceptype;

    (ec,il) = AArch64.ExceptionClass(exceptype, target_el);
    iss  = except.syndrome.iss;
    iss2 = except.syndrome.iss2;

    // IL is not valid for Data Abort exceptions without valid instruction syndrome information
    if ec IN {0x24,0x25} && iss<24> == '0' then
        il = '1';

    ESR_EL[target_el] = (Zeros(8)  :   // <63:56>
                         iss2      :   // <55:32>
                         ec<5:0>   :   // <31:26>
                         il        :   // <25>
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                          iss);         // <24:0>

    if exceptype IN {
        Exception_InstructionAbort,
        Exception_PCAlignment,
        Exception_DataAbort,
        Exception_Watchpoint
    } then
        FAR_EL[target_el] = except.vaddress;
    else
        FAR_EL[target_el] = bits(64) UNKNOWN;

    if except.ipavalid then
        HPFAR_EL2<43:4> = except.ipaddress<51:12>;
        if IsSecureEL2Enabled() && CurrentSecurityState() == SS_Secure then
            HPFAR_EL2.NS = except.NS;
        else
            HPFAR_EL2.NS = '0';
    elsif target_el == EL2 then
        HPFAR_EL2<43:4> = bits(40) UNKNOWN;

    return;
```

G1.1.2.21 AArch64.ResetControlRegisters

```
// AArch64.ResetControlRegisters()
// ===============================
// Resets System registers and memory-mapped control registers that have architecturally-defined
// reset values to those values.

AArch64.ResetControlRegisters(boolean cold_reset);
```

G1.1.2.22 AArch64.TakeReset

```
// AArch64.TakeReset()
// ===================
// Reset into AArch64 state

AArch64.TakeReset(boolean cold_reset)
    assert HaveAArch64();

    // Enter the highest implemented Exception level in AArch64 state
    PSTATE.nRW = '0';
    if HaveEL(EL3) then
        PSTATE.EL = EL3;
    elsif HaveEL(EL2) then
        PSTATE.EL = EL2;
    else
        PSTATE.EL = EL1;

    // Reset System registers and other system components
    AArch64.ResetControlRegisters(cold_reset);

    // Reset all other PSTATE fields
    PSTATE.SP = '1';                    // Select stack pointer
    PSTATE.<D,A,I,F>  = '1111';         // All asynchronous exceptions masked
    PSTATE.SS = '0';                    // Clear software step bit
    PSTATE.DIT = '0';                   // PSTATE.DIT is reset to 0 when resetting into AArch64
    if IsFeatureImplemented(FEAT_SSBS) then
        PSTATE.SSBS = bit IMPLEMENTATION_DEFINED "PSTATE.SSBS bit at reset";
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    PSTATE.IL = '0';                    // Clear Illegal Execution state bit

    // All registers, bits and fields not reset by the above pseudocode or by the BranchTo() call
    // below are UNKNOWN bitstrings after reset. In particular, the return information registers
    // ELR_ELx and SPSR_ELx have UNKNOWN values, so that it
    // is impossible to return from a reset in an architecturally defined way.
    AArch64.ResetGeneralRegisters();
    AArch64.ResetSpecialRegisters();
    ResetExternalDebugRegisters(cold_reset);

    bits(64) rv;                        // IMPLEMENTATION DEFINED reset vector

    rv = RVBAR_EL2;

    // The reset vector must be correctly aligned
    constant AddressSize pamax = AArch64.PAMax();
    assert IsZero(rv<63:pamax>) && IsZero(rv<1:0>);

    constant boolean branch_conditional = FALSE;
    EDPRSR.R = '0';                  // Leaving Reset State.
    BranchTo(rv, BranchType_RESET, branch_conditional);
```

G1.1.2.23 AArch64.FPTrappedException

```
// AArch64.FPTrappedException()
// ============================

AArch64.FPTrappedException(boolean is_ase, bits(8) accumulated_exceptions)
    except = ExceptionSyndrome(Exception_FPTrappedException);
    if is_ase then
        if boolean IMPLEMENTATION_DEFINED "vector instructions set TFV to 1" then
            except.syndrome.iss<23> = '1';                          // TFV
        else
            except.syndrome.iss<23> = '0';                          // TFV
    else
        except.syndrome.iss<23> = '1';                              // TFV
    except.syndrome.iss<10:8> = bits(3) UNKNOWN;                    // VECITR
    if except.syndrome.iss<23> == '1' then
        except.syndrome.iss<7,4:0> = accumulated_exceptions<7,4:0>; // IDF,IXF,UFF,OFF,DZF,IOF
    else
        except.syndrome.iss<7,4:0> = bits(6) UNKNOWN;

    route_to_el2 = EL2Enabled() && HCR_EL2.TGE == '1';

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, except, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```

G1.1.2.24 AArch64.CallHypervisor

```
// AArch64.CallHypervisor()
// ========================
// Performs a HVC call
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
AArch64.CallHypervisor(bits(16) immediate)
    assert HaveEL(EL2);

    SSAdvance();
    constant bits(64) preferred_exception_return = NextInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception_HypervisorCall);
    except.syndrome.iss<15:0> = immediate;

    if PSTATE.EL == EL3 then
        AArch64.TakeException(EL3, except, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
```

G1.1.2.25 AArch64.CallSecureMonitor

```
// AArch64.CallSecureMonitor()
// ===========================

AArch64.CallSecureMonitor(bits(16) immediate)
    assert HaveEL(EL3) && !ELUsingAArch32(EL3);
    HSAdvance();
    SSAdvance();
    constant bits(64) preferred_exception_return = NextInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception_MonitorCall);
    except.syndrome.iss<15:0> = immediate;
    AArch64.TakeException(EL3, except, preferred_exception_return, vect_offset);
```

G1.1.2.26 AArch64.CallSupervisor

```
// AArch64.CallSupervisor()
// ========================
// Calls the Supervisor

AArch64.CallSupervisor(bits(16) immediate)
    SSAdvance();
    route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';

    constant bits(64) preferred_exception_return = NextInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception_SupervisorCall);
    except.syndrome.iss<15:0> = immediate;
    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, except, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```

G1.1.2.27 AArch64.TakeException

```
// AArch64.TakeException()
// =======================
// Take an exception to an Exception level using AArch64.
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
AArch64.TakeException(bits(2) target_el, ExceptionRecord exception_in,
                      bits(64) preferred_exception_return, integer vect_offset_in)
    assert HaveEL(target_el) && !ELUsingAArch32(target_el) && UInt(target_el) >= UInt(PSTATE.EL);
    if Halted() then
        AArch64.TakeExceptionInDebugState(target_el, exception_in);
        return;
    ExceptionRecord except = exception_in;
    boolean sync_errors;
    boolean iesb_req;
    if IsFeatureImplemented(FEAT_IESB) then
        sync_errors = SCTLR_EL[target_el].IESB == '1';
        if sync_errors && InsertIESBBeforeException(target_el) then
            SynchronizeErrors();
            if except.exceptype != Exception_SError then
                iesb_req = FALSE;
                sync_errors = FALSE;
                TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);
    else
        sync_errors = FALSE;

    // Synchronize the context, including Instruction Fetch Barrier effect
    SynchronizeContext();

    // If coming from AArch32 state, the top parts of the X[] registers might be set to zero
    from_32 = UsingAArch32();
    if from_32 then AArch64.MaybeZeroRegisterUppers();

    integer vect_offset = vect_offset_in;
    if UInt(target_el) > UInt(PSTATE.EL) then
        boolean lower_32;
        if target_el == EL3 then
            if EL2Enabled() then
                lower_32 = ELUsingAArch32(EL2);
            else
                lower_32 = ELUsingAArch32(EL1);
        else
            lower_32 = ELUsingAArch32(target_el - 1);
        vect_offset = vect_offset + (if lower_32 then 0x600 else 0x400);

    elsif PSTATE.SP == '1' then
        vect_offset = vect_offset + 0x200;

    assert vect_offset < 2048;
    constant bits(64) vect_base = VBAR_EL[target_el]<63:11>:Zeros(11);
    constant bits(64) target_vector = vect_base + vect_offset;

    bits(64) spsr = GetPSRFromPSTATE(AArch64_NonDebugState, 64);

    if ! except.exceptype IN {Exception_IRQ, Exception_FIQ} then
        AArch64.ReportException(except, target_el);

    PSTATE.EL = target_el;
    PSTATE.nRW = '0';
    PSTATE.SP = '1';

    SPSR_ELx[] = spsr;
    ELR_ELx[] = preferred_exception_return;
    PSTATE.SS = '0';
    if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = SCTLR_ELx[].DSSBS;
    PSTATE.<D,A,I,F> = '1111';
    PSTATE.IL = '0';
    if IsFeatureImplemented(FEAT_UAO) then PSTATE.UAO = '0';
    if IsFeatureImplemented(FEAT_PAN) && PSTATE.EL == EL1 && SCTLR_EL1.SPAN == '0' then
        PSTATE.PAN = '1';
    constant boolean branch_conditional = FALSE;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    BranchTo(target_vector, BranchType_EXCEPTION, branch_conditional);
    CheckExceptionCatch(TRUE);                  // Check for debug event on exception entry

    if sync_errors then
        SynchronizeErrors();
        iesb_req = TRUE;
        TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);

    EndOfInstruction();
```

G1.1.2.28 AArch64.AdvSIMDFPAccessTrap

```
// AArch64.AdvSIMDFPAccessTrap()
// ============================
// Trapped access to Advanced SIMD or FP registers due to CPACR.

AArch64.AdvSIMDFPAccessTrap(bits(2) target_el)
    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    ExceptionRecord except;
    vect_offset = 0x0;

    route_to_el2 = (target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1');

    if route_to_el2 then
        except = ExceptionSyndrome(Exception_Uncategorized);
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        except = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
        except.syndrome.iss<24:20> = ConditionSyndrome();
        AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);

    return;
```

G1.1.2.29 AArch64.CheckFPAdvSIMDEnabled

```
// AArch64.CheckFPAdvSIMDEnabled()
// ===============================

AArch64.CheckFPAdvSIMDEnabled()
    AArch64.CheckFPEnabled();
```

G1.1.2.30 AArch64.CheckFPAdvSIMDTrap

```
// AArch64.CheckFPAdvSIMDTrap()
// ============================
// Check against CPTR_EL2 and CPTR_EL3.

AArch64.CheckFPAdvSIMDTrap()

    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
        // Check if access disabled in CPTR_EL2
        if CPTR_EL2.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL2);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.1.2.31 AArch64.CheckFPEnabled

```
// AArch64.CheckFPEnabled()
// ========================
// Check against CPACR

AArch64.CheckFPEnabled()
    if PSTATE.EL IN {EL0, EL1} then
        // Check if access disabled in CPACR_EL1
        boolean disabled;
        case CPACR_EL1.FPEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0;
            when '11' disabled = FALSE;
        if disabled then AArch64.AdvSIMDFPAccessTrap(EL1);

    AArch64.CheckFPAdvSIMDTrap();               // Also check against CPTR_EL2 and CPTR_EL3
```

G1.1.2.32 AArch64.CheckForSMCUndefOrTrap

```
// AArch64.CheckForSMCUndefOrTrap()
// ================================
// Check for UNDEFINED or trap on SMC instruction

AArch64.CheckForSMCUndefOrTrap(bits(16) imm)
    if PSTATE.EL == EL0 then UNDEFINED;
    route_to_el2 = FALSE;
    if !HaveEL(EL3) || PSTATE.EL == EL0 then
        UNDEFINED;
    route_to_el2 = PSTATE.EL == EL1 && EL2Enabled() && HCR_EL2.TSC == '1';
    if route_to_el2 then
        constant bits(64) preferred_exception_return = ThisInstrAddr(64);
        vect_offset = 0x0;
        except = ExceptionSyndrome(Exception_MonitorCall);
        except.syndrome.iss<15:0> = imm;
        except.trappedsyscallinst = TRUE;
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
```

G1.1.2.33 AArch64.CheckForWFxTrap

```
// AArch64.CheckForWFxTrap()
// =========================
// Checks for a trap on a WFE or WFI instruction.

(boolean, bits(2)) AArch64.CheckForWFxTrap(WFxType wfxtype)
    constant boolean is_wfe = wfxtype == WFxType_WFE;
    bits(2) target_el;
    boolean trap = FALSE;

    if !trap && PSTATE.EL == EL0 then
        // Check for traps described by the OS.
        trap      = (if is_wfe then SCTLR_ELx[].nTWE else SCTLR_ELx[].nTWI) == '0';
        target_el = EL1;

    if !trap && PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        // Check for traps described by the Hypervisor.
        trap      = (if is_wfe then HCR_EL2.TWE else HCR_EL2.TWI) == '1';
        target_el = EL2;

    return (trap, target_el);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.1.2.34 AArch64.CheckIllegalState

```
// AArch64.CheckIllegalState()
// ===========================
// Check PSTATE.IL bit and generate Illegal Execution state exception if set.

AArch64.CheckIllegalState()
    if PSTATE.IL == '1' then
        route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';

        constant bits(64) preferred_exception_return = ThisInstrAddr(64);
        vect_offset = 0x0;

        except = ExceptionSyndrome(Exception_IllegalState);

        if UInt(PSTATE.EL) > UInt(EL1) then
            AArch64.TakeException(PSTATE.EL, except, preferred_exception_return, vect_offset);
        elsif route_to_el2 then
            AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
        else
            AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```

G1.1.2.35 AArch64.MonitorModeTrap

```
// AArch64.MonitorModeTrap()
// =========================
// Trapped use of Monitor mode features in a Secure EL1 AArch32 mode

AArch64.MonitorModeTrap()
    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception_Uncategorized);

    if IsSecureEL2Enabled() then
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
    AArch64.TakeException(EL3, except, preferred_exception_return, vect_offset);
```

G1.1.2.36 AArch64.SystemAccessTrap

```
// AArch64.SystemAccessTrap()
// ==========================
// Trapped access to AArch64 System register or system instruction.

AArch64.SystemAccessTrap(bits(2) target_el, integer ec)
    assert HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = AArch64.SystemAccessTrapSyndrome(ThisInstr(), ec);
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

G1.1.2.37 AArch64.SystemAccessTrapSyndrome

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// AArch64.SystemAccessTrapSyndrome()
// ==================================
// Returns the syndrome information for traps on AArch64 MSR/MRS instructions.

ExceptionRecord AArch64.SystemAccessTrapSyndrome(bits(32) instr_in, integer ec)
    ExceptionRecord except;
    bits(32) instr = instr_in;
    case ec of
        when 0x0                        // Trapped access due to unknown reason.
            except = ExceptionSyndrome(Exception_Uncategorized);
        when 0x7                        // Trapped access to SVE, Advance SIMD&FP System register.
            except = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
            except.syndrome.iss<24:20> = ConditionSyndrome();
        when 0x18                       // Trapped access to System register or system instruction.
            except = ExceptionSyndrome(Exception_SystemRegisterTrap);
            instr = ThisInstr();
            except.syndrome.iss<21:20> = instr<20:19>; // Op0
            except.syndrome.iss<19:17> = instr<7:5>;   // Op2
            except.syndrome.iss<16:14> = instr<18:16>; // Op1
            except.syndrome.iss<13:10> = instr<15:12>; // CRn
            except.syndrome.iss<9:5>   = instr<4:0>;   // Rt
            except.syndrome.iss<4:1>   = instr<11:8>;  // CRm
            except.syndrome.iss<0>     = instr<21>;    // Direction
        otherwise
            Unreachable();

    return except;
```

G1.1.2.38 AArch64.Undefined

```
// AArch64.Undefined()
// ===================

AArch64.Undefined()

    route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';
    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception_Uncategorized);

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, except, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```

G1.1.2.39 AArch64.WFxTrap

```
// AArch64.WFxTrap()
// =================
// Generate an exception for a trapped WFE or WFI instruction.

AArch64.WFxTrap(WFxType wfxtype, bits(2) target_el)
    assert UInt(target_el) > UInt(PSTATE.EL);
    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    constant integer vect_offset = 0x0;
    ExceptionRecord except = ExceptionSyndrome(Exception_WFxTrap);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    except.syndrome.iss<24:20> = ConditionSyndrome();
    case wfxtype of
        when WFxType_WFI
            except.syndrome.iss<0> = '0';
        when WFxType_WFE
            except.syndrome.iss<0> = '1';

    if target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1' then
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

G1.1.3 aarch64/functions

This section includes the following pseudocode functions:

G1.1.3.1 AArch64.FaultSyndrome

```
// AArch64.FaultSyndrome()
// =======================
// Creates an exception syndrome value and updates the virtual address for Abort and Watchpoint
// exceptions taken to an Exception level using AArch64.

IssType AArch64.FaultSyndrome(Exception exceptype, FaultRecord fault, bits(2) target_el)
    assert fault.statuscode != Fault_None;

    IssType isstype;
    isstype.iss  = Zeros(25);
    isstype.iss2 = Zeros(24);

    constant boolean d_side = exceptype IN {Exception_DataAbort, Exception_Watchpoint};
    if IsFeatureImplemented(FEAT_RAS) && fault.statuscode == Fault_SyncExternal then
        constant ErrorState errstate = PEErrorState(fault);
        isstype.iss<12:11> = AArch64.EncodeSyncErrorSyndrome(errstate);  // SET

    if d_side then
        if exceptype == Exception_Watchpoint then
            isstype.iss<23:0> = WatchpointRelatedSyndrome(fault);

        if AArch64.InstructionSyndromeValid(fault) then
            isstype.iss<24:14> = LSInstructionSyndrome();

        if fault.accessdesc.acctype IN {AccessType_DC, AccessType_IC, AccessType_AT} then
            isstype.iss<8> = '1';

        if fault.accessdesc.acctype IN {AccessType_DC, AccessType_IC, AccessType_AT} then
            isstype.iss<6> = '1';
        elsif fault.statuscode IN {Fault_HWUpdateAccessFlag, Fault_Exclusive} then
            isstype.iss<6> = bit UNKNOWN;
        elsif fault.accessdesc.atomicop && IsExternalAbort(fault) then
            isstype.iss<6> = bit UNKNOWN;
        else
            isstype.iss<6> = if fault.write then '1' else '0';

    if IsExternalAbort(fault) then isstype.iss<9> = fault.extflag;
    isstype.iss<7> = if fault.s2fs1walk then '1' else '0';
    isstype.iss<5:0> = EncodeLDFSC(fault.statuscode, fault.level);

    return isstype;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.1.3.2 AArch64.InstructionSyndromeValid

```
// AArch64.InstructionSyndromeValid()
// ==================================
// Returns TRUE if ESR_ELx.ISV is '1' for the given Fault.

boolean AArch64.InstructionSyndromeValid(FaultRecord fault)
    if !fault.accessdesc.lowestaddress then
        return FALSE;

    if IsSecondStage(fault) && !fault.s2fs1walk then
        return (!IsExternalSyncAbort(fault) ||
               (!IsFeatureImplemented(FEAT_RAS) && IsExternalAbortOnWalk(fault) &&
                  boolean IMPLEMENTATION_DEFINED "ISV on second stage translation table walk"));

    return FALSE;
```

G1.1.3.3 AArch64.AT

```
// AArch64.AT()
// ============
// Perform address translation as per AT instructions.

AArch64.AT(bits(64) address, TranslationStage stage, bits(2) el_in, ATAccess ataccess)
    bits(2) el = el_in;
    constant SecurityState ss = SS_Secure;

    accdesc = CreateAccDescAT(ss, el, ataccess);
    aligned = TRUE;

    FaultRecord fault = NoFault(accdesc, address);
    Regime regime;
    if stage == TranslationStage_12 then
        regime = Regime_EL10;
    else
        regime = TranslationRegime(el);

    AddressDescriptor addrdesc;
    if AArch64.IsStage1VMSA(regime) then
        constant integer size = 1;
        (fault, addrdesc) = AArch64.S1Translate(fault, regime, address, size, aligned, accdesc);
    else
        (fault, addrdesc) = AArch64.S1Validate(fault, regime, address, aligned, accdesc);

    if stage == TranslationStage_12 && fault.statuscode == Fault_None then
        constant boolean s1aarch64 = TRUE;
        if regime == Regime_EL10 then
            (fault, addrdesc) = AArch64.S2Validate(fault, addrdesc, aligned, accdesc);

    is_ATS1Ex = stage != TranslationStage_12;
    if fault.statuscode != Fault_None then
        addrdesc = CreateFaultyAddressDescriptor(fault);
        // Take an exception on:
        // * A Synchronous External abort occurs on translation table walk
        // * A stage 2 fault occurs on a stage 1 walk
        if IsExternalAbort(fault) || (PSTATE.EL == EL1 && fault.s2fs1walk) then
            PAR_EL1 = bits(64) UNKNOWN;
            AArch64.Abort(addrdesc.fault);

    AArch64.EncodePAR(regime, el, addrdesc);
    return;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.1.3.4 AArch64.EncodePAR

```
// AArch64.EncodePAR()
// ===================
// Encode PAR register with result of translation.

AArch64.EncodePAR(Regime regime, bits(2) el, AddressDescriptor addrdesc)
    paspace = addrdesc.paddress.paspace;
    PAR_EL1 = Zeros(64);

    if !IsFault(addrdesc) then
        PAR_EL1.F = '0';
        PAR_EL1<11> = '1'; // RES1
        if SecurityStateForRegime(regime) == SS_Secure then
            PAR_EL1.NS = if paspace == PAS_Secure then '0' else '1';
        else
            PAR_EL1.NS = bit UNKNOWN;
        PAR_EL1.SH   = ReportedPARShareability(PAREncodeShareability(addrdesc.memattrs));
        PAR_EL1.PA   = addrdesc.paddress.address<51:12>;
        PAR_EL1.ATTR = ReportedPARAttrs(EncodePARAttrs(addrdesc.memattrs));
        PAR_EL1<10> = bit IMPLEMENTATION_DEFINED "Non-Faulting PAR";
    else
        PAR_EL1.F   = '1';
        PAR_EL1.FST = AArch64.PARFaultStatus(addrdesc.fault);
        PAR_EL1.PTW = if addrdesc.fault.s2fs1walk then '1' else '0';
        PAR_EL1.S   = if addrdesc.fault.secondstage then '1' else '0';
        PAR_EL1<11> = '1'; // RES1
        PAR_EL1<63:48> = bits(16) IMPLEMENTATION_DEFINED "Faulting PAR";
    return;
```

G1.1.3.5 AArch64.PARFaultStatus

```
// AArch64.PARFaultStatus()
// ========================
// Fault status field decoding of 64-bit PAR.

bits(6) AArch64.PARFaultStatus(FaultRecord fault)
    bits(6) fst;

    if fault.statuscode == Fault_Domain then
        // Report Domain fault
        assert fault.level IN {1,2};
        fst<1:0> = if fault.level == 1 then '01' else '10';
        fst<5:2> = '1111';
    else
        fst = EncodeLDFSC(fault.statuscode, fault.level);
    return fst;
```

G1.1.3.6 GetPAR_EL1_F

```
// GetPAR_EL1_F()
// ==============
// Query the PAR_EL1.F field.

bit GetPAR_EL1_F()
    bit F;

    F = PAR_EL1.F;
    return F;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.1.3.7 MemBarrierOp

```
// MemBarrierOp
// ============
// Memory barrier instruction types.

enumeration MemBarrierOp   {MemBarrierOp_DSB,         // Data Synchronization Barrier
                            MemBarrierOp_DMB,         // Data Memory Barrier
                            MemBarrierOp_ISB,         // Instruction Synchronization Barrier
                            MemBarrierOp_SSBB,        // Speculative Synchronization Barrier to VA
                            MemBarrierOp_PSSBB,       // Speculative Synchronization Barrier to PA
                            MemBarrierOp_SB           // Speculation Barrier
                           };
```

G1.1.3.8 BFXPreferred

```
// BFXPreferred()
// ==============
//
// Return TRUE if UBFX or SBFX is the preferred disassembly of a
// UBFM or SBFM bitfield instruction. Must exclude more specific
// aliases UBFIZ, SBFIZ, UXT[BH], SXT[BHW], LSL, LSR and ASR.

boolean BFXPreferred(bit sf, bit uns, bits(6) imms, bits(6) immr)

    // must not match UBFIZ/SBFIX alias
    if UInt(imms) < UInt(immr) then
        return FALSE;

    // must not match LSR/ASR/LSL alias (imms == 31 or 63)
    if imms == sf:'11111' then
        return FALSE;

    // must not match UXTx/SXTx alias
    if immr == '000000' then
        // must not match 32-bit UXT[BH] or SXT[BH]
        if sf == '0' && imms IN {'000111', '001111'} then
            return FALSE;
        // must not match 64-bit SXT[BHW]
        if sf:uns == '10' && imms IN {'000111', '001111', '011111'} then
            return FALSE;

    // must be UBFX/SBFX alias
    return TRUE;
```

G1.1.3.9 AltDecodeBitMasks

```
// AltDecodeBitMasks()
// ===================
// Alternative but logically equivalent implementation of DecodeBitMasks() that
// uses simpler primitives to compute tmask and wmask.

(bits(M), bits(M)) AltDecodeBitMasks(bit immN, bits(6) imms, bits(6) immr,
                                     boolean immediate, integer M)
    bits(64) tmask, wmask;
    bits(6) tmask_and, wmask_and;
    bits(6) tmask_or, wmask_or;
    bits(6) levels;

    // Compute log2 of element size
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// 2^len must be in range [2, M]
if immN:NOT(imms) == '000000x' then UNDEFINED;
constant integer len = HighestSetBitNZ(immN:NOT(imms));
assert 2 <= (2^len) && (2^len) <= M;

// Determine s, r and s - r parameters
levels = ZeroExtend(Ones(len), 6);

// For logical immediates an all-ones value of s is reserved
// since it would generate a useless all-ones result (many times)
if immediate && (imms AND levels) == levels then UNDEFINED;

s = UInt(imms AND levels);
r = UInt(immr AND levels);
diff = s - r;    // 6-bit subtract with borrow

// Compute "top mask"
tmask_and = diff<5:0> OR NOT(levels);
tmask_or  = diff<5:0> AND levels;

tmask = Ones(64);
tmask = ((tmask
          AND Replicate(Replicate(tmask_and<0>, 1) : Ones(1), 32))
           OR  Replicate(Zeros(1) : Replicate(tmask_or<0>, 1), 32));
// tmask = Replicate(tmask_and<0> : '1', 32);
tmask = ((tmask
          AND Replicate(Replicate(tmask_and<1>, 2) : Ones(2), 16))
           OR  Replicate(Zeros(2) : Replicate(tmask_or<1>, 2), 16));
tmask = ((tmask
          AND Replicate(Replicate(tmask_and<2>, 4) : Ones(4), 8))
           OR  Replicate(Zeros(4) : Replicate(tmask_or<2>, 4), 8));
tmask = ((tmask
          AND Replicate(Replicate(tmask_and<3>, 8) : Ones(8), 4))
           OR  Replicate(Zeros(8) : Replicate(tmask_or<3>, 8), 4));
tmask = ((tmask
          AND Replicate(Replicate(tmask_and<4>, 16) : Ones(16), 2))
           OR  Replicate(Zeros(16) : Replicate(tmask_or<4>, 16), 2));
tmask = ((tmask
          AND Replicate(Replicate(tmask_and<5>, 32) : Ones(32), 1))
           OR  Replicate(Zeros(32) : Replicate(tmask_or<5>, 32), 1));

// Compute "wraparound mask"
wmask_and = immr OR NOT(levels);
wmask_or  = immr AND levels;

wmask = Zeros(64);
wmask = ((wmask
          AND Replicate(Ones(1) : Replicate(wmask_and<0>, 1), 32))
           OR  Replicate(Replicate(wmask_or<0>, 1) : Zeros(1), 32));
// wmask = Replicate(wmask_or<0> : '0', 32);
wmask = ((wmask
          AND Replicate(Ones(2) : Replicate(wmask_and<1>, 2), 16))
           OR  Replicate(Replicate(wmask_or<1>, 2) : Zeros(2), 16));
wmask = ((wmask
          AND Replicate(Ones(4) : Replicate(wmask_and<2>, 4), 8))
           OR  Replicate(Replicate(wmask_or<2>, 4) : Zeros(4), 8));
wmask = ((wmask
          AND Replicate(Ones(8) : Replicate(wmask_and<3>, 8), 4))
           OR  Replicate(Replicate(wmask_or<3>, 8) : Zeros(8), 4));
wmask = ((wmask
          AND Replicate(Ones(16) : Replicate(wmask_and<4>, 16), 2))
           OR  Replicate(Replicate(wmask_or<4>, 16) : Zeros(16), 2));
wmask = ((wmask
          AND Replicate(Ones(32) : Replicate(wmask_and<5>, 32), 1))
           OR  Replicate(Replicate(wmask_or<5>, 32) : Zeros(32), 1));
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if diff<6> != '0' then // borrow from s - r
        wmask = wmask AND tmask;
    else
        wmask = wmask OR tmask;

    return (wmask<M-1:0>, tmask<M-1:0>);
```

G1.1.3.10 DecodeBitMasks

```
// DecodeBitMasks()
// ================
// Decode AArch64 bitfield and logical immediate masks which use a similar encoding structure

(bits(M), bits(M)) DecodeBitMasks(bit immN, bits(6) imms, bits(6) immr,
                                  boolean immediate, integer M)
    bits(M) tmask, wmask;
    bits(6) levels;

    // Compute log2 of element size
    // 2^len must be in range [2, M]
    if immN:NOT(imms) == '000000x' then UNDEFINED;
    constant integer len = HighestSetBitNZ(immN:NOT(imms));
    assert 2 <= (2^len) && (2^len) <= M;

    // Determine s, r and s - r parameters
    levels = ZeroExtend(Ones(len), 6);

    // For logical immediates an all-ones value of s is reserved
    // since it would generate a useless all-ones result (many times)
    if immediate && (imms AND levels) == levels then UNDEFINED;

    constant integer s = UInt(imms AND levels);
    constant integer r = UInt(immr AND levels);
    constant integer diff = s - r;    // 6-bit subtract with borrow

    constant integer esize = 1 << len;
    constant integer d = UInt(diff<len-1:0>);
    welem = ZeroExtend(Ones(s + 1), esize);
    telem = ZeroExtend(Ones(d + 1), esize);
    wmask = Replicate(ROR(welem, r), M DIV esize);
    tmask = Replicate(telem, M DIV esize);
    return (wmask, tmask);
```

G1.1.3.11 AArch64.DataMemZero

```
// AArch64.DataMemZero()
// =====================
// Write Zero to data memory.

AArch64.DataMemZero(bits(64) regval, bits(64) vaddress, AccessDescriptor accdesc_in, integer size)
    AccessDescriptor accdesc = accdesc_in;

    constant boolean aligned = TRUE;
    AddressDescriptor memaddrdesc = AArch64.TranslateAddress(vaddress, accdesc, aligned, size);

    if IsFault(memaddrdesc) then
        if !IsDebugException(memaddrdesc.fault) then
            memaddrdesc.fault.vaddress = regval;
        AArch64.Abort(memaddrdesc.fault);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    for i = 0 to size-1
        memstatus = PhysMemWrite(memaddrdesc, 1, accdesc, Zeros(8));
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc);

        memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1;
    return;
```

G1.1.3.12 CompareOp

```
// CompareOp
// =========
// Vector compare instruction types.

enumeration CompareOp   {CompareOp_GT, CompareOp_GE, CompareOp_EQ,
                         CompareOp_LE, CompareOp_LT};
```

G1.1.3.13 CountOp

```
// CountOp
// =======
// Bit counting instruction types.

enumeration CountOp     {CountOp_CLZ, CountOp_CLS, CountOp_CNT};
```

G1.1.3.14 AArch64.CanTrapDC

```
// AArch64.CanTrapDC()
// ===================
// Determines whether the execution of the DC instruction can be trapped.

boolean AArch64.CanTrapDC(CacheType cachetype, CacheOp cacheop, CacheOpScope opscope)
    return (!AArch64.TreatDCAsNOP(cachetype, cacheop, opscope) || boolean IMPLEMENTATION_DEFINED
            "When DC is treated as NOP, data cache maintenance operations are trapped");
```

G1.1.3.15 AArch64.DC

```
// AArch64.DC()
// ============
// Perform Data Cache Operation.

AArch64.DC(bits(64) regval, CacheType cachetype, CacheOp cacheop, CacheOpScope opscope_in)
    CacheOpScope opscope = opscope_in;
    CacheRecord cache;

    cache.acctype   = AccessType_DC;
    cache.cachetype = cachetype;
    cache.cacheop   = cacheop;
    cache.opscope   = opscope;

    if opscope == CacheOpScope_SetWay then
        ss = SecurityStateAtEL(PSTATE.EL);
        cache.cpas = CPASAtSecurityState(ss);
        cache.shareability = Shareability_NSH;
        (cache.setnum, cache.waynum, cache.level) = DecodeSW(regval, cachetype);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if (cacheop == CacheOp_Invalidate && PSTATE.EL == EL1 && EL2Enabled() &&
          (HCR_EL2.SWIO == '1' || HCR_EL2.<DC,VM> != '00')) then
            cache.cacheop = CacheOp_CleanInvalidate;

        CACHE_OP(cache);
        return;

    if EL2Enabled() then
        if PSTATE.EL IN {EL0, EL1} then
            cache.is_vmid_valid = TRUE;
            cache.vmid          = VMID[];
        else
            cache.is_vmid_valid = FALSE;
    else
        cache.is_vmid_valid = FALSE;

    if PSTATE.EL == EL0 then
        cache.is_asid_valid = TRUE;
        cache.asid          = ASID[];
    else
        cache.is_asid_valid = FALSE;

    if (opscope == CacheOpScope_PoDP &&
          boolean IMPLEMENTATION_DEFINED "Memory system does not support PoDP") then
        opscope = CacheOpScope_PoP;
    if (opscope == CacheOpScope_PoP &&
          boolean IMPLEMENTATION_DEFINED "Memory system does not support PoP") then
        opscope = CacheOpScope_PoC;
    vaddress = regval;

    integer size = 0;        // by default no watchpoint address
    if cacheop == CacheOp_Invalidate then
        size = DataCacheWatchpointSize();
        vaddress = Align(regval, size);

    cache.vaddress  = vaddress;
    constant boolean aligned = TRUE;
    constant AccessDescriptor accdesc = CreateAccDescDC(cache);
    AddressDescriptor memaddrdesc = AArch64.TranslateAddress(vaddress, accdesc,
                                                             aligned, size);
    if IsFault(memaddrdesc) then
        memaddrdesc.fault.vaddress = regval;
        AArch64.Abort(memaddrdesc.fault);

    cache.paddress   = memaddrdesc.paddress;
    cache.cpas       = CPASAtPAS(memaddrdesc.paddress.paspace);

    if (opscope IN {
                    CacheOpScope_PoDP,
                    CacheOpScope_PoP,
                    CacheOpScope_PoC,
                    CacheOpScope_PoU
                }) then
        cache.shareability = memaddrdesc.memattrs.shareability;
    else
        cache.shareability = Shareability_NSH;

    if (cacheop == CacheOp_Invalidate && PSTATE.EL == EL1 && EL2Enabled() &&
            HCR_EL2.<DC,VM> != '00') then
        cache.cacheop = CacheOp_CleanInvalidate;

    CACHE_OP(cache);
    return;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.1.3.16 AArch64.MemZero

```
// AArch64.MemZero()
// =================

AArch64.MemZero(bits(64) regval, CacheType cachetype)
    constant integer size = 4*(2^(UInt(DCZID_EL0.BS)));
    assert size <= MAX_ZERO_BLOCK_SIZE;
    constant bits(64) vaddress = Align(regval, size);

    constant AccessDescriptor accdesc = CreateAccDescDCZero();

    AArch64.DataMemZero(regval, vaddress, accdesc, size);
    return;
```

G1.1.3.17 AArch64.TreatDCAsNOP

```
// AArch64.TreatDCAsNOP()
// ======================
// Determines whether the execution of the DC instruction is treated as a NOP.

boolean AArch64.TreatDCAsNOP(CacheType cachetype, CacheOp cacheop, CacheOpScope opscope)
    // DC to PoU: IMPLEMENTATION DEFINED - treated as NOP if LoUU and LoUIS are 0
    if opscope == CacheOpScope_PoU && CLIDR_EL1.LoUU == '000' && CLIDR_EL1.LoUIS == '000' then
        return boolean IMPLEMENTATION_DEFINED "DC to PoU is treated as a NOP";

    // DC to PoC: IMPLEMENTATION DEFINED - treated as NOP if LoC is 0
    if opscope == CacheOpScope_PoC && CLIDR_EL1.LoC == '000' then
        return boolean IMPLEMENTATION_DEFINED "DC to PoC is treated as a NOP";

    // DC to PoP: IMPLEMENTATION DEFINED - treated as NOP if LoC is 0 and PoP unsupported
    if (opscope == CacheOpScope_PoP && CLIDR_EL1.LoC == '000' &&
          !(boolean IMPLEMENTATION_DEFINED "Memory system supports PoP")) then
        return boolean IMPLEMENTATION_DEFINED "DC to PoP is treated as a NOP";

    // DC to PoDP: IMPLEMENTATION DEFINED - treated as NOP if LoC is 0 and PoP/PoDP unsupported
    if (opscope == CacheOpScope_PoDP && CLIDR_EL1.LoC == '000' &&
          !(boolean IMPLEMENTATION_DEFINED "Memory system supports PoP") &&
          !(boolean IMPLEMENTATION_DEFINED "Memory system supports PoDP")) then
        return boolean IMPLEMENTATION_DEFINED "DC to PoDP is treated as a NOP";

    return FALSE;
```

G1.1.3.18 MemZero

```
// MemZero block size
// ==================

constant integer MAX_ZERO_BLOCK_SIZE = 2048;
```

G1.1.3.19 AArch64.ExceptionReturn

```
// AArch64.ExceptionReturn()
// =========================

AArch64.ExceptionReturn(bits(64) new_pc_in, bits(64) spsr)
    bits(64) new_pc = new_pc_in;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if IsFeatureImplemented(FEAT_IESB) then
        sync_errors = SCTLR_ELx[].IESB == '1';
        if sync_errors then
            SynchronizeErrors();
            iesb_req = TRUE;
            TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);

    SynchronizeContext();

    // Attempts to change to an illegal state will invoke the Illegal Execution state mechanism
    constant bits(2) source_el = PSTATE.EL;
    constant boolean illegal_psr_state = IllegalExceptionReturn(spsr);
    SetPSTATEFromPSR(spsr, illegal_psr_state);
    ClearExclusiveLocal(ProcessorID());
    SendEventLocal();

    if illegal_psr_state && spsr<4> == '1' then
        // If the exception return is illegal, PC[63:32,1:0] are UNKNOWN
        new_pc<63:32> = bits(32) UNKNOWN;
        new_pc<1:0> = bits(2) UNKNOWN;
    else                                    // Return to AArch64
        // ELR_ELx[63:56] might include a tag
        new_pc = AArch64.BranchAddr(new_pc, PSTATE.EL);

    if UsingAArch32() then
        // 32 most significant bits are ignored.
        constant boolean branch_conditional = FALSE;
        BranchTo(new_pc<31:0>, BranchType_ERET, branch_conditional);
    else
        BranchToAddr(new_pc, BranchType_ERET);

    CheckExceptionCatch(FALSE);             // Check for debug event on exception return
```

G1.1.3.20 AArch64.ExclusiveMonitorsPass

```
// AArch64.ExclusiveMonitorsPass()
// ===============================
// Return TRUE if the Exclusives monitors for the current PE include all of the addresses
// associated with the virtual address region of size bytes starting at address.
// The immediately following memory write must be to the same addresses.

// It is IMPLEMENTATION DEFINED whether the detection of memory aborts happens
// before or after the check on the local Exclusives monitor. As a result a failure
// of the local monitor can occur on some implementations even if the memory
// access would give an memory abort.

boolean AArch64.ExclusiveMonitorsPass(bits(64) address, integer size,
                                      AccessDescriptor accdesc)
    constant boolean aligned = IsAligned(address, size);

    if !aligned && AArch64.UnalignedAccessFaults(accdesc, address, size) then
        constant FaultRecord fault = AlignmentFault(accdesc, address);
        AArch64.Abort(fault);

    if !AArch64.IsExclusiveVA(address, ProcessorID(), size) then
        return FALSE;

    memaddrdesc = AArch64.TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    passed = IsExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);
    ClearExclusiveLocal(ProcessorID());

    if passed && memaddrdesc.memattrs.shareability != Shareability_NSH then
        passed = IsExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

    return passed;
```

G1.1.3.21 AArch64.IsExclusiveVA

```
// AArch64.IsExclusiveVA()
// =======================
// An optional IMPLEMENTATION DEFINED test for an exclusive access to a virtual
// address region of size bytes starting at address.
//
// It is permitted (but not required) for this function to return FALSE and
// cause a store exclusive to fail if the virtual address region is not
// totally included within the region recorded by MarkExclusiveVA().
//
// It is always safe to return TRUE which will check the physical address only.

boolean AArch64.IsExclusiveVA(bits(64) address, integer processorid, integer size);
```

G1.1.3.22 AArch64.MarkExclusiveVA

```
// AArch64.MarkExclusiveVA()
// =========================
// Optionally record an exclusive access to the virtual address region of size bytes
// starting at address for processorid.

AArch64.MarkExclusiveVA(bits(64) address, integer processorid, integer size);
```

G1.1.3.23 AArch64.SetExclusiveMonitors

```
// AArch64.SetExclusiveMonitors()
// ==============================
// Sets the Exclusives monitors for the current PE to record the addresses associated
// with the virtual address region of size bytes starting at address.

AArch64.SetExclusiveMonitors(bits(64) address, integer size,
                             AccessDescriptor accdesc)
    constant boolean aligned = IsAligned(address, size);

    if !aligned && AArch64.UnalignedAccessFaults(accdesc, address, size) then
        constant FaultRecord fault = AlignmentFault(accdesc, address);
        AArch64.Abort(fault);

    memaddrdesc = AArch64.TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return;

    if memaddrdesc.memattrs.shareability != Shareability_NSH then
        MarkExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

    MarkExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    AArch64.MarkExclusiveVA(address, ProcessorID(), size);
```

G1.1.3.24 DecodeRegExtend

```
// DecodeRegExtend()
// =================
// Decode a register extension option

ExtendType DecodeRegExtend(bits(3) op)
    case op of
        when '000' return ExtendType_UXTB;
        when '001' return ExtendType_UXTH;
        when '010' return ExtendType_UXTW;
        when '011' return ExtendType_UXTX;
        when '100' return ExtendType_SXTB;
        when '101' return ExtendType_SXTH;
        when '110' return ExtendType_SXTW;
        when '111' return ExtendType_SXTX;
```

G1.1.3.25 ExtendReg

```
// ExtendReg()
// ===========
// Perform a register extension and shift

bits(N) ExtendReg(integer reg, ExtendType exttype, integer shift, integer N)
    assert shift >= 0 && shift <= 4;
    constant bits(N) val = X[reg, N];
    boolean unsigned;
    ESize len;

    case exttype of
        when ExtendType_SXTB unsigned = FALSE; len = 8;
        when ExtendType_SXTH unsigned = FALSE; len = 16;
        when ExtendType_SXTW unsigned = FALSE; len = 32;
        when ExtendType_SXTX unsigned = FALSE; len = 64;
        when ExtendType_UXTB unsigned = TRUE;  len = 8;
        when ExtendType_UXTH unsigned = TRUE;  len = 16;
        when ExtendType_UXTW unsigned = TRUE;  len = 32;
        when ExtendType_UXTX unsigned = TRUE;  len = 64;

    // Sign or zero extend bottom LEN bits of register and shift left by SHIFT
    constant nbits = Min(len, N);
    constant bits(N) extval = Extend(val<nbits-1:0>, N, unsigned);
    return LSL(extval, shift);
```

G1.1.3.26 ExtendType

```
// ExtendType
// ==========
// AArch64 register extend and shift.

enumeration ExtendType  {ExtendType_SXTB, ExtendType_SXTH, ExtendType_SXTW, ExtendType_SXTX,
                         ExtendType_UXTB, ExtendType_UXTH, ExtendType_UXTW, ExtendType_UXTX};
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.1.3.27 FPConvOp

```
// FPConvOp
// ========
// Floating-point convert/move instruction types.

enumeration FPConvOp    {FPConvOp_CVT_FtoI, FPConvOp_CVT_ItoF,
                         FPConvOp_MOV_FtoI, FPConvOp_MOV_ItoF,
                         FPConvOp_CVT_FtoI_JS
};
```

G1.1.3.28 FPMaxMinOp

```
// FPMaxMinOp
// ==========
// Floating-point min/max instruction types.

enumeration FPMaxMinOp  {FPMaxMinOp_MAX, FPMaxMinOp_MIN,
                         FPMaxMinOp_MAXNUM, FPMaxMinOp_MINNUM};
```

G1.1.3.29 FPUnaryOp

```
// FPUnaryOp
// =========
// Floating-point unary instruction types.

enumeration FPUnaryOp   {FPUnaryOp_ABS, FPUnaryOp_MOV,
                         FPUnaryOp_NEG, FPUnaryOp_SQRT};
```

G1.1.3.30 FPRSqrtStepFused

```
// FPRSqrtStepFused()
// ==================

bits(N) FPRSqrtStepFused(bits(N) op1_in, bits(N) op2, FPCR_Type fpcr_in)
    assert N IN {16, 32, 64};
    FPCR_Type fpcr = fpcr_in;
    bits(N) result;
    bits(N) op1 = op1_in;
    boolean done;
    op1 = FPNeg(op1, fpcr);

    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    constant FPRounding rounding = FPRoundingMode(fpcr);

    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);

        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPOnePointFive('0', N);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2, N);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            // Fully fused multiply-add and halve
            result_value = (3.0 + (value1 * value2)) / 2.0;
            if result_value == 0.0 then
                // Sign of exact zero result depends on rounding mode
                sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(sign, N);
            else
                result = FPRound(result_value, fpcr, rounding, N);

    return result;
```

G1.1.3.31 FPRecipStepFused

```
// FPRecipStepFused()
// ==================

bits(N) FPRecipStepFused(bits(N) op1_in, bits(N) op2, FPCR_Type fpcr_in)
    assert N IN {16, 32, 64};
    FPCR_Type fpcr = fpcr_in;
    bits(N) op1 = op1_in;
    bits(N) result;
    boolean done;
    op1 = FPNeg(op1, fpcr);

    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    constant FPRounding rounding = FPRoundingMode(fpcr);

    if !done then
        inf1  = (type1 == FPType_Infinity);
        inf2  = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);

        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPTwo('0', N);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2, N);
        else
            // Fully fused multiply-add
            result_value = 2.0 + (value1 * value2);
            if result_value == 0.0 then
                // Sign of exact zero result depends on rounding mode
                sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(sign, N);
            else
                result = FPRound(result_value, fpcr, rounding, N);

    return result;
```

G1.1.3.32 AArch64.CanTrapIC

```
// AArch64.CanTrapIC()
// ===================
// Determines whether the execution of the IC instruction can be trapped.

boolean AArch64.CanTrapIC(CacheType cachetype, CacheOp cacheop, CacheOpScope opscope)
    return (!AArch64.TreatICAsNOP(cachetype, cacheop, opscope) ||boolean IMPLEMENTATION_DEFINED
            "When IC is treated as NOP, data cache maintenance operations are trapped");
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.1.3.33 AArch64.IC

```
// AArch64.IC()
// ============
// Perform Instruction Cache Operation.

AArch64.IC(CacheOpScope opscope)
    regval = bits(64) UNKNOWN;
    AArch64.IC(regval, opscope);

// AArch64.IC()
// ============
// Perform Instruction Cache Operation.

AArch64.IC(bits(64) regval, CacheOpScope opscope)
    CacheRecord cache;

    cache.acctype   = AccessType_IC;
    cache.cachetype = CacheType_Instruction;
    cache.cacheop   = CacheOp_Invalidate;
    cache.opscope   = opscope;

    if opscope IN {CacheOpScope_ALLU,
                   CacheOpScope_ALLUIS} then
        ss = SecurityStateAtEL(PSTATE.EL);
        cache.cpas = CPASAtSecurityState(ss);

        case opscope of
            when CacheOpScope_ALLU
                cache.shareability = Shareability_NSH;
            when CacheOpScope_ALLUIS
                cache.shareability = Shareability_ISH;
            otherwise
                Unreachable();

        cache.regval = regval;
        CACHE_OP(cache);
    else
        assert opscope == CacheOpScope_PoU;

        if EL2Enabled() then
            if PSTATE.EL IN {EL0, EL1} then
                cache.is_vmid_valid = TRUE;
                cache.vmid          = VMID[];
            else
                cache.is_vmid_valid = FALSE;
        else
            cache.is_vmid_valid = FALSE;

        if PSTATE.EL == EL0 then
            cache.is_asid_valid = TRUE;
            cache.asid          = ASID[];
        else
            cache.is_asid_valid = FALSE;

        constant bits(64) vaddress = regval;
        cache.vaddress     = regval;

        constant AccessDescriptor accdesc = CreateAccDescIC(cache);
        constant boolean aligned = TRUE;
        constant integer size = 0;
        AddressDescriptor memaddrdesc = AArch64.TranslateAddress(vaddress, accdesc, aligned, size);

        if IsFault(memaddrdesc) then
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            memaddrdesc.fault.vaddress = regval;
            AArch64.Abort(memaddrdesc.fault);

        cache.cpas         = CPASAtPAS(memaddrdesc.paddress.paspace);
        cache.paddress     = memaddrdesc.paddress;
        cache.shareability = memaddrdesc.memattrs.shareability;
        if memaddrdesc.memattrs.shareability == Shareability_OSH then
            cache.shareability = Shareability_ISH;
        CACHE_OP(cache);
    return;
```

G1.1.3.34 AArch64.TreatICAsNOP

```
// AArch64.TreatICAsNOP()
// ======================
// Determines whether the execution of the IC instruction is treated as a NOP.

boolean AArch64.TreatICAsNOP(CacheType cachetype, CacheOp cacheop, CacheOpScope opscope)
    if CTR_EL0.DIC == '1' then
        return boolean IMPLEMENTATION_DEFINED "IC is treated as NOP";

    return FALSE;
```

G1.1.3.35 ImmediateOp

```
// ImmediateOp
// ===========
// Vector logical immediate instruction types.

enumeration ImmediateOp {ImmediateOp_MOVI, ImmediateOp_MVNI,
                         ImmediateOp_ORR, ImmediateOp_BIC};
```

G1.1.3.36 LogicalOp

```
// LogicalOp
// =========
// Logical instruction types.

enumeration LogicalOp   {LogicalOp_AND, LogicalOp_EOR, LogicalOp_ORR};
```

G1.1.3.37 AArch64.IsUnprivAccessPriv

```
// AArch64.IsUnprivAccessPriv()
// ============================
// Returns TRUE if an unprivileged access is privileged, and FALSE otherwise.

boolean AArch64.IsUnprivAccessPriv()
    boolean privileged;

    case PSTATE.EL of
        when EL0
            privileged = FALSE;
        when EL1
            privileged = FALSE;
        when EL2
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            privileged = TRUE;
        when EL3
            privileged = TRUE;

    if IsFeatureImplemented(FEAT_UAO) && PSTATE.UAO == '1' then
        privileged = PSTATE.EL != EL0;

    return privileged;
```

G1.1.3.38 AArch64.MemSingle

```
// AArch64.MemSingle - non-assignment (read) form
// ==============================================
// Perform an atomic, little-endian read of 'size' bytes.

bits(size*8) AArch64.MemSingle[bits(64) address, integer size,
                               AccessDescriptor accdesc, boolean aligned]
    bits(size*8) value;
    AddressDescriptor memaddrdesc;
    PhysMemRetStatus memstatus;

    (value, memaddrdesc, memstatus) = AArch64.MemSingleRead(address, size, accdesc, aligned);

    // Check for a fault from translation or the output of translation.
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);

    // Check for external aborts.
    if IsFault(memstatus) then
        HandleExternalAbort(memstatus, accdesc.write, memaddrdesc, size, accdesc);

    return value;

// AArch64.MemSingle - assignment (write) form
// ===========================================
// Perform an atomic, little-endian write of 'size' bytes.

AArch64.MemSingle[bits(64) address, integer size, AccessDescriptor accdesc,
                  boolean aligned] = bits(size*8) value
    AddressDescriptor memaddrdesc;
    PhysMemRetStatus memstatus;

    (memaddrdesc, memstatus) = AArch64.MemSingleWrite(address, size, accdesc, aligned, value);

    // Check for a fault from translation or the output of translation.
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);

    // Check for external aborts.
    if IsFault(memstatus) then
        HandleExternalWriteAbort(memstatus, memaddrdesc, size, accdesc);

    return;
```

G1.1.3.39 AArch64.MemSingleRead

```
// AArch64.MemSingleRead()
// =======================
// Perform an atomic, little-endian read of 'size' bytes.
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
(bits(size*8), AddressDescriptor, PhysMemRetStatus) AArch64.MemSingleRead(bits(64) address,
                    integer size,
                    AccessDescriptor accdesc_in,
                    boolean aligned)
    assert size IN {1, 2, 4, 8, 16, 32};
    bits(size*8) value = bits(size*8) UNKNOWN;
    PhysMemRetStatus memstatus = PhysMemRetStatus UNKNOWN;
    memstatus.statuscode = Fault_None;
    AccessDescriptor accdesc = accdesc_in;
    if IsFeatureImplemented(FEAT_LSE2) then
        constant integer quantity = MemSingleGranule();
        assert AllInAlignedQuantity(address, size, quantity);
    else
        assert IsAligned(address, size);

    AddressDescriptor memaddrdesc;
    memaddrdesc = AArch64.TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return (value, memaddrdesc, memstatus);

    // Memory array access
    boolean atomic;
    if IsWBShareable(memaddrdesc.memattrs) then
        atomic = TRUE;
    elsif accdesc.exclusive then
        atomic = TRUE;
    elsif aligned then
        atomic = !accdesc.ispair;
    else
        // Misaligned accesses within MemSingleGranule() byte aligned memory but
        // not Normal Cacheable Writeback are Atomic
        atomic = boolean IMPLEMENTATION_DEFINED "FEAT_LSE2: access is atomic";

    if atomic then
        (memstatus, value) = PhysMemRead(memaddrdesc, size, accdesc);

    elsif aligned && accdesc.ispair then
        assert size IN {4,  8,  16};
        constant integer halfsize = size DIV 2;
        bits(halfsize * 8) lowhalf, highhalf;
        (memstatus, lowhalf) = PhysMemRead(memaddrdesc, halfsize, accdesc);

        if !IsFault(memstatus) then
            memaddrdesc.paddress.address = memaddrdesc.paddress.address + halfsize;
            memaddrdesc.vaddress         = memaddrdesc.vaddress + halfsize;
            (memstatus, highhalf) = PhysMemRead(memaddrdesc, halfsize, accdesc);

        if !IsFault(memstatus) then
            value = highhalf:lowhalf;
    else
        for i = 0 to size-1
            if !IsFault(memstatus) then     // Do not continue past a fault
                (memstatus, Elem[value, i, 8]) = PhysMemRead(memaddrdesc, 1, accdesc);

            if !IsFault(memstatus) then
                memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1;
                memaddrdesc.vaddress         = memaddrdesc.vaddress + 1;

    if IsFault(memstatus) then
        return (value, memaddrdesc, memstatus);

    if accdesc.acctype == AccessType_IFETCH then
        memaddrdesc.fault = AArch64.CheckDebug(address, accdesc, size);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    return (value, memaddrdesc, memstatus);
```

G1.1.3.40 AArch64.MemSingleWrite

```
// AArch64.MemSingleWrite()
// ========================
// Perform an atomic, little-endian write of 'size' bytes.

(AddressDescriptor, PhysMemRetStatus) AArch64.MemSingleWrite(bits(64) address,
                                                             integer size,
                                                             AccessDescriptor accdesc_in,
                                                             boolean aligned, bits(size*8) value)
    assert size IN {1, 2, 4, 8, 16, 32};
    AccessDescriptor accdesc = accdesc_in;
    if IsFeatureImplemented(FEAT_LSE2) then
        constant integer quantity = MemSingleGranule();
        assert AllInAlignedQuantity(address, size, quantity);
    else
        assert IsAligned(address, size);

    AddressDescriptor memaddrdesc;
    PhysMemRetStatus memstatus = PhysMemRetStatus UNKNOWN;
    memaddrdesc = AArch64.TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return (memaddrdesc, memstatus);

    // Effect on exclusives
    if memaddrdesc.mematttrs.shareability != Shareability_NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

    boolean atomic;
    if IsWBShareable(memaddrdesc.mematttrs) then
        atomic = TRUE;
    elsif accdesc.exclusive then
        atomic = TRUE;
    elsif aligned then
        atomic = !accdesc.ispair;
    else
        // Misaligned accesses within MemSingleGranule() byte aligned memory but
        // not Normal Cacheable Writeback are Atomic
        atomic = boolean IMPLEMENTATION_DEFINED "FEAT_LSE2: access is atomic";
    if atomic then
        memstatus = PhysMemWrite(memaddrdesc, size, accdesc, value);
        if IsFault(memstatus) then
            return (memaddrdesc, memstatus);

    elsif aligned && accdesc.ispair then
        assert size IN {8,  16};
        constant integer halfsize = size DIV 2;
        bits(halfsize*8) lowhalf, highhalf;
        (highhalf, lowhalf) = (value<halfsize*8+:halfsize*8>, value<0+:halfsize*8>);

        memstatus = PhysMemWrite(memaddrdesc, halfsize, accdesc, lowhalf);
        if IsFault(memstatus) then
            return (memaddrdesc, memstatus);

        memaddrdesc.paddress.address = memaddrdesc.paddress.address + halfsize;
        memaddrdesc.vaddress         = memaddrdesc.vaddress + halfsize;
        memstatus = PhysMemWrite(memaddrdesc, halfsize, accdesc, highhalf);
        if IsFault(memstatus) then
            return (memaddrdesc, memstatus);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        for i = 0 to size-1
            memstatus = PhysMemWrite(memaddrdesc, 1, accdesc, Elem[value, i, 8]);
            if IsFault(memstatus) then
                return (memaddrdesc, memstatus);
            memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1;
            memaddrdesc.vaddress         = memaddrdesc.vaddress + 1;

    return (memaddrdesc, memstatus);
```

G1.1.3.41 AArch64.UnalignedAccessFaults

```
// AArch64.UnalignedAccessFaults()
// ===============================
// Determine whether the unaligned access generates an Alignment fault

boolean AArch64.UnalignedAccessFaults(AccessDescriptor accdesc, bits(64) address, integer size)
    if AlignmentEnforced() then
        return TRUE;
    elsif (accdesc.exclusive || accdesc.atomicop)  then
        constant integer quantity = MemSingleGranule();
        return (!IsFeatureImplemented(FEAT_LSE2) ||
                  !AllInAlignedQuantity(address, size, quantity));
    elsif (accdesc.acqsc || accdesc.acqpc || accdesc.relsc) then
        // If nAA is 0, the address accessed by each register in the pair
        // must lie within a single 16-byte aligned block
        if accdesc.ispair then
            return  (SCTLR_ELx[].nAA == '0' &&
                       (!AllInAlignedQuantity(address, size DIV 2, 16) ||
                        !AllInAlignedQuantity(address + size DIV 2, size DIV 2, 16)));
        else
            return  (SCTLR_ELx[].nAA == '0' &&
                       !AllInAlignedQuantity(address, size, 16));
    else
        return FALSE;
```

G1.1.3.42 AllInAlignedQuantity

```
// AllInAlignedQuantity()
// ======================
// Returns TRUE if all accessed bytes are within one aligned quantity, FALSE otherwise.

boolean AllInAlignedQuantity(bits(64) address, integer size, integer alignment)
    return (Align(address+(size-1), alignment) ==
            Align(address, alignment));
```

G1.1.3.43 CASCompare

```
// CASCompare()
// ============
// Performs a comparison for CAS

(bits(N), boolean, bits(N)) CASCompare(bits(N) oldvalue, bits(N) comparevalue, bits(N) newvalue)
    bits(N) memresult;
    boolean cmpfail;
    bits(N) regresult = oldvalue;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if oldvalue == comparevalue then
        cmpfail = FALSE;
        memresult = newvalue;
        if !ConstrainUnpredictableBool(Unpredictable_CASRETURNOLDVALUE) then
            regresult = comparevalue;
    else
        cmpfail = TRUE;
        memresult = oldvalue;

    return (memresult, cmpfail, regresult);
```

G1.1.3.44 CheckSPAlignment

```
// CheckSPAlignment()
// ==================
// Check correct stack pointer alignment for AArch64 state.

CheckSPAlignment()
    constant bits(64) sp = SP[64];
    boolean stack_align_check;
    if PSTATE.EL == EL0 then
        stack_align_check = (SCTLR_ELx[].SA0 != '0');
    else
        stack_align_check = (SCTLR_ELx[].SA != '0');

    if stack_align_check && sp != Align(sp, 16) then
        AArch64.SPAlignmentFault();

    return;
```

G1.1.3.45 Mem

```
// Mem - non-assignment (read) form
// ================================
// Perform a read of 'size' bytes. The access byte order is reversed for a big-endian access.
// Instruction fetches would call AArch64.MemSingle directly.

bits(size*8) Mem[bits(64) address, integer size,
                 AccessDescriptor accdesc_in]
    assert size IN {1,  2,  4,  8,  16,  32};
    AccessDescriptor accdesc = accdesc_in;
    bits(size * 8) value;
    // Check alignment on size of element accessed, not overall access size
    constant integer alignment = (if accdesc.ispair && !accdesc.exclusive then size DIV 2
                                                                          else size);
    boolean aligned   = IsAligned(address, alignment);
    constant integer quantity = MemSingleGranule();

    if !aligned && AArch64.UnalignedAccessFaults(accdesc, address, size) then
        constant FaultRecord fault = AlignmentFault(accdesc, address);
        AArch64.Abort(fault);
    if accdesc.acctype == AccessType_ASIMD && size == 16 && IsAligned(address, 8) then
        // If 128-bit SIMD&FP ordered access are treated as a pair of
        // 64-bit single-copy atomic accesses, then these single copy atomic
        // access can be observed in any order.
        constant integer halfsize = size DIV 2;
        constant bits(64) highaddress = AddressIncrement(address, halfsize, accdesc);
        bits(halfsize * 8) lowhalf, highhalf;
        lowhalf = AArch64.MemSingle[address, halfsize, accdesc, aligned];
        highhalf = AArch64.MemSingle[highaddress, halfsize, accdesc, aligned];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            value = highhalf:lowhalf;
        elsif accdesc.acctype == AccessType_ASIMD && size == 32 && IsAligned(address, 8) then
            // If a 32 byte LDP (SIMD&FP) access is not aligned to 32 bytes but aligned to
            // 8 bytes, it is treated as four 8 byte single-copy atomic accesses.
            accdesc.ispair = FALSE;
            aligned        = TRUE;
            for i = 0 to 3
                constant bits(64) blockaddress = AddressIncrement(address, i*8, accdesc);
                value<64*i+:64> = AArch64.MemSingle[blockaddress, 8, accdesc, aligned];
        elsif (IsFeatureImplemented(FEAT_LSE2) &&
               AllInAlignedQuantity(address, size, quantity)) then
            value = AArch64.MemSingle[address, size, accdesc, aligned];
        elsif aligned && accdesc.ispair then
            // Either: an aligned pair access, OR
            // an SVE predicated load of a 128-bit element that is 64-bit aligned,
            // which is treated as two 64-bit single-copy atomic accesses.
            assert size > 1;
            accdesc.ispair = FALSE;
            constant integer halfsize = size DIV 2;
            constant bits(64) highaddress = AddressIncrement(address, halfsize, accdesc);
            bits(halfsize * 8) lowhalf, highhalf;
            lowhalf  = AArch64.MemSingle[address, halfsize, accdesc, aligned];
            highhalf = AArch64.MemSingle[highaddress, halfsize, accdesc, aligned];
            value = highhalf:lowhalf;
        elsif aligned then
            value = AArch64.MemSingle[address, size, accdesc, aligned];
        else
            assert size > 1;
            value<7:0> = AArch64.MemSingle[address, 1, accdesc, aligned];

            // For subsequent bytes, if they cross to a new translation page which assigns
            // Device memory type, it is CONSTRAINED UNPREDICTABLE whether an unaligned access
            // will generate an Alignment Fault.
            c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
            assert c IN {Constraint_FAULT, Constraint_NONE};
            if c == Constraint_NONE then aligned = TRUE;

            for i = 1 to size-1
                Elem[value, i, 8] = AArch64.MemSingle[address+i, 1, accdesc, aligned];

        if BigEndian(accdesc.acctype) then
            value = BigEndianReverse(value);

        return value;

    // Mem - assignment (write) form
    // =============================
    // Perform a write of 'size' bytes. The byte order is reversed for a big-endian access.

    Mem[bits(64) address, integer size,
        AccessDescriptor accdesc_in] = bits(size*8) value_in
        bits(size*8) value = value_in;
        AccessDescriptor accdesc = accdesc_in;

        // Check alignment on size of element accessed, not overall access size
        constant integer alignment = (if accdesc.ispair && !accdesc.exclusive then size DIV 2
                                                                                else size);
        boolean aligned   = IsAligned(address, alignment);
        constant integer quantity = MemSingleGranule();

        if !aligned && AArch64.UnalignedAccessFaults(accdesc, address, size) then
            constant FaultRecord fault = AlignmentFault(accdesc, address);
            AArch64.Abort(fault);

        if BigEndian(accdesc.acctype) then
            value = BigEndianReverse(value);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if accdesc.acctype == AccessType_ASIMD && size == 16 && IsAligned(address, 8) then
        constant integer halfsize = size DIV 2;
        bits(halfsize*8) lowhalf, highhalf;
        // 128-bit SIMD&FP stores are treated as a pair of 64-bit single-copy atomic accesses
        // 64-bit aligned.
        (highhalf, lowhalf) = (value<halfsize*8+:halfsize*8>, value<0+:halfsize*8>);
        constant bits(64) highaddress = AddressIncrement(address, halfsize, accdesc);
        AArch64.MemSingle[address, halfsize, accdesc, aligned] = lowhalf;
        AArch64.MemSingle[highaddress, halfsize, accdesc, aligned] = highhalf;
    elsif accdesc.acctype == AccessType_ASIMD && size == 32 && IsAligned(address, 8) then
        // If a 32 byte STP (SIMD&FP) access is not aligned to 32 bytes but aligned to
        // 8 bytes, it is treated as four 8 byte single-copy atomic accesses.
        accdesc.ispair = FALSE;
        aligned        = TRUE;
        for i = 0 to 3
            constant bits(64) blockaddress = AddressIncrement(address, i*8, accdesc);
            AArch64.MemSingle[blockaddress, 8, accdesc, aligned] = value<64*i+:64>;
    elsif (IsFeatureImplemented(FEAT_LSE2) &&
           AllInAlignedQuantity(address, size, quantity)) then
        AArch64.MemSingle[address, size, accdesc, aligned] = value;
    elsif aligned && accdesc.ispair then
        // Either: an aligned pair access, OR
        // an SVE predicated load of a 128-bit element that is 64-bit aligned,
        // which is treated as two 64-bit single-copy atomic accesses.
        assert size > 1;
        constant integer halfsize = size DIV 2;
        bits(halfsize*8) lowhalf, highhalf;
        (highhalf, lowhalf) = (value<halfsize*8+:halfsize*8>, value<0+:halfsize*8>);
        accdesc.ispair = FALSE;
        constant bits(64) highaddress = AddressIncrement(address, halfsize, accdesc);
        AArch64.MemSingle[address,     halfsize, accdesc, aligned] = lowhalf;
        AArch64.MemSingle[highaddress, halfsize, accdesc, aligned] = highhalf;
    elsif aligned then
        AArch64.MemSingle[address, size, accdesc, aligned] = value;
    else
        assert size > 1;
        AArch64.MemSingle[address, 1, accdesc, aligned] = value<7:0>;

        // For subsequent bytes, if they cross to a new translation page which assigns
        // Device memory type, it is CONSTRAINED UNPREDICTABLE whether an unaligned access
        // will generate an Alignment Fault.

        c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
        assert c IN {Constraint_FAULT, Constraint_NONE};
        if c == Constraint_NONE then aligned = TRUE;

        for i = 1 to size-1
            constant bits(64) byteaddress = AddressIncrement(address, i, accdesc);
            AArch64.MemSingle[byteaddress, 1, accdesc, aligned] = Elem[value, i, 8];
    return;
```

G1.1.3.46 MemAtomic

```
// MemAtomic()
// ===========
// Performs load and store memory operations for a given virtual address.

bits(size) MemAtomic(bits(64) address, bits(size) cmpoperand, bits(size) operand,
                     AccessDescriptor accdesc_in)
    assert accdesc_in.atomicop;

    constant integer bytes = size DIV 8;
    assert bytes IN {1,  2,  4,  8,  16};
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    bits(size) newvalue;
    bits(size) oldvalue;
    AccessDescriptor accdesc = accdesc_in;
    constant boolean aligned = IsAligned(address, bytes);

    if !aligned && AArch64.UnalignedAccessFaults(accdesc, address, bytes) then
        constant FaultRecord fault = AlignmentFault(accdesc, address);
        AArch64.Abort(fault);

    // MMU or MPU lookup
    constant AddressDescriptor memaddrdesc = AArch64.TranslateAddress(address, accdesc,
                                                                      aligned, bytes);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);

    if (!IsWBShareable(memaddrdesc.memattrs) &&
            ConstrainUnpredictableBool(Unpredictable_Atomic_NOP)) then
        return bits(size) UNKNOWN;

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != Shareability_NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), bytes);

    // All observers in the shareability domain observe the following load and store atomically.
    PhysMemRetStatus memstatus;
    (memstatus, oldvalue) = PhysMemRead(memaddrdesc, bytes, accdesc);
    // Depending on the memory type of the physical address, the access might generate
    // either a synchronous External abort or an SError exception
    // among other CONSTRAINED UNPREDICTABLE choices.

    if IsFault(memstatus) then
        HandleExternalReadAbort(memstatus, memaddrdesc, bytes, accdesc);
    if BigEndian(accdesc.acctype) then
        oldvalue = BigEndianReverse(oldvalue);

    boolean cmpfail = FALSE;
    bits(size) retvalue = oldvalue;
    (newvalue, cmpfail, retvalue) = MemAtomicInt(accdesc.modop, oldvalue, operand, cmpoperand);

    constant boolean requirewrite = (!cmpfail ||
                                     ConstrainUnpredictableBool(Unpredictable_WRITEFAILEDCAS));

    if requirewrite then
        if BigEndian(accdesc.acctype) then
            newvalue = BigEndianReverse(newvalue);
        memstatus = PhysMemWrite(memaddrdesc, bytes, accdesc, newvalue);
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, bytes, accdesc);

    // Load operations return the old (pre-operation) value.
    // Compare and Swap operations return the old (pre-operation) value. For a successful CAS,
    // this might be the value from the compare operand or from memory.
    return retvalue;
```

G1.1.3.47 MemAtomicInt

```
// MemAtomicInt()
// ==============
// Performs Integer Atomic operation
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
(bits(N), boolean, bits(N)) MemAtomicInt(MemAtomicOp modop, bits(N) op1, bits(N) op2, bits(N) cmpop)
    bits(N) result;
    boolean cmpfail = FALSE;
    bits(N) retvalue = op1;

    case modop of
        when MemAtomicOp_ADD      result = op1 + op2;
        when MemAtomicOp_BIC      result = op1 AND NOT(op2);
        when MemAtomicOp_EOR      result = op1 EOR op2;
        when MemAtomicOp_ORR      result = op1 OR op2;
        when MemAtomicOp_SMAX     result = Max(SInt(op1), SInt(op2))<N-1:0>;
        when MemAtomicOp_SMIN     result = Min(SInt(op1), SInt(op2))<N-1:0>;
        when MemAtomicOp_UMAX     result = Max(UInt(op1), UInt(op2))<N-1:0>;
        when MemAtomicOp_UMIN     result = Min(UInt(op1), UInt(op2))<N-1:0>;
        when MemAtomicOp_SWP      result = op2;
        when MemAtomicOp_CAS      (result, cmpfail, retvalue) = CASCompare(op1, cmpop, op2);
    return (result, cmpfail, retvalue);
```

G1.1.3.48 MemSingleGranule

```
// MemSingleGranule()
// ==================
// When FEAT_LSE2 is implemented, for some memory accesses if all bytes
// of the accesses are within 16-byte quantity aligned to 16-bytes and
// satisfy additional requirements - then the access is guaranteed to
// be single copy atomic.
// However, when the accesses do not all lie within such a boundary, it
// is CONSTRAINED UNPREDICTABLE if the access is single copy atomic.
// In the pseudocode, this CONSTRAINED UNPREDICTABLE aspect is modeled via
// MemSingleGranule() which is IMPLEMENTATION DEFINED and, is at least 16 bytes
// and at most 4096 bytes.
// This is a limitation of the pseudocode.

integer MemSingleGranule()
    size = integer IMPLEMENTATION_DEFINED "Aligned quantity for atomic access";
    // access is assumed to be within 4096 byte aligned quantity to
    // avoid multiple translations for a single copy atomic access.
    assert (size >= 16) && (size <= 4096);
    return size;
```

G1.1.3.49 MoveWideOp

```
// MoveWideOp
// ==========
// Move wide 16-bit immediate instruction types.

enumeration MoveWideOp  {MoveWideOp_N, MoveWideOp_Z, MoveWideOp_K};
```

G1.1.3.50 MoveWidePreferred

```
// MoveWidePreferred()
// ===================
//
// Return TRUE if a bitmask immediate encoding would generate an immediate
// value that could also be represented by a single MOVZ or MOVN instruction.
// Used as a condition for the preferred MOV<-ORR alias.

boolean MoveWidePreferred(bit sf, bit immN, bits(6) imms, bits(6) immr)
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    constant integer s = UInt(imms);
    constant integer r = UInt(immr);
    constant integer width = if sf == '1' then 64 else 32;

    // element size must equal total immediate size
    if sf == '1' && (immN:imms) != '1xxxxxx' then
        return FALSE;
    if sf == '0' && (immN:imms) != '00xxxxx' then
        return FALSE;

    // for MOVZ must contain no more than 16 ones
    if s < 16 then
        // ones must not span halfword boundary when rotated
        return (-r MOD 16) <= (15 - s);

    // for MOVN must contain no more than 16 zeros
    if s >= width - 15 then
        // zeros must not span halfword boundary when rotated
        return (r MOD 16) <= (s - (width - 15));

    return FALSE;
```

G1.1.3.51 AddPAC

```
// AddPAC()
// ========
// Calculates the pointer authentication code for a 64-bit quantity and then
// inserts that into pointer authentication code field of that 64-bit quantity.

bits(64) AddPAC(bits(64) ptr, bits(64) modifier, bits(128) K, boolean data)
    bits(64) PAC;
    bits(64) result;
    bits(64) ext_ptr;
    bits(64) extfield;
    bit selbit;
    bit bit55;
    constant boolean tbi = EffectiveTBI(ptr, !data, PSTATE.EL) == '1';
    constant integer top_bit = if tbi then 55 else 63;

    // If tagged pointers are in use for a regime with two TTBRs, use bit<55> of
    // the pointer to select between upper and lower ranges, and preserve this.
    // This handles the awkward case where there is apparently no correct choice between
    // the upper and lower address range - ie an addr of 1xxxxxxx0... with TBI0=0 and TBI1=1
    // and 0xxxxxxx1 with TBI1=0 and TBI0=1:
    // This include EL1/EL0 in both VMSA and PMSA context.
    if PSTATE.EL == EL1 || PSTATE.EL == EL0 then
        assert S1TranslationRegime() == EL1;
        if S1TranslationRegime() == EL1 then
            // EL1 translation regime registers
            if data then
                if TCR_EL1.TBI1 == '1' || TCR_EL1.TBI0 == '1' then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
            else
                if ((TCR_EL1.TBI1 == '1' && TCR_EL1.TBID1 == '0') ||
                    (TCR_EL1.TBI0 == '1' && TCR_EL1.TBID0 == '0')) then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
    else selbit = if tbi then ptr<55> else ptr<63>;

    if IsFeatureImplemented(FEAT_PAuth2) && IsFeatureImplemented(FEAT_CONSTPACFIELD) then
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        selbit = ptr<55>;
    constant AddressSize bottom_PAC_bit = CalculateBottomPACBit(selbit);

    extfield = Replicate(selbit, 64);

    // Compute the pointer authentication code for a ptr with good extension bits
    if tbi then
        if bottom_PAC_bit <= 55 then
            ext_ptr = (ptr<63:56> :
                   extfield<55:bottom_PAC_bit> : ptr<bottom_PAC_bit-1:0>);
        else
            ext_ptr = ptr<63:56> : ptr<55:0>;
    else
        ext_ptr =  extfield<63:bottom_PAC_bit> : ptr<bottom_PAC_bit-1:0>;

    PAC = ComputePAC(ext_ptr, modifier, K<127:64>, K<63:0>);

    if !IsFeatureImplemented(FEAT_PAuth2) then
        // If FEAT_PAuth2 is not implemented, the PAC is corrupted if the pointer does not have
        // a canonical VA.
        if !IsZero(ptr<top_bit:bottom_PAC_bit>) && !IsOnes(ptr<top_bit:bottom_PAC_bit>) then
            PAC<top_bit-1> = NOT(PAC<top_bit-1>);

    // Preserve the determination between upper and lower address at bit<55> and insert PAC into
    // bits that are not used for the address or the tag(s).
    if !IsFeatureImplemented(FEAT_PAuth2) then
        assert (bottom_PAC_bit <= 52);
        if tbi then
            result = ptr<63:56>:selbit:PAC<54:bottom_PAC_bit>:ptr<bottom_PAC_bit-1:0>;
        else
            result = PAC<63:56>:selbit:PAC<54:bottom_PAC_bit>:ptr<bottom_PAC_bit-1:0>;
    else
        bit55 = selbit;
        if tbi then
            if bottom_PAC_bit < 55 then
                result = (ptr<63:56>                                  : bit55 :
                          (ptr<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>) :
                          ptr<bottom_PAC_bit-1:0>);
            else
                result = (ptr<63:56> : bit55 : ptr<54:0>);
        else
            if bottom_PAC_bit < 55 then
                result = ((ptr<63:56> EOR PAC<63:56>)                 : bit55 :
                          (ptr<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>) :
                          ptr<bottom_PAC_bit-1:0>);
            else
                result = ((ptr<63:56> EOR PAC<63:56>)                 : bit55 :
                           ptr<54:0>);
    return result;
```

G1.1.3.52 AddPACDA

```
// AddPACDA()
// ==========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of x, y and the
// APDAKey_EL1.

bits(64) AddPACDA(bits(64) x, bits(64) y)
    constant bits(128) APDAKey_EL1 = APDAKeyHi_EL1<63:0> : APDAKeyLo_EL1<63:0>;
    if !IsAPDAKeyEnabled() then
        return x;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        return AddPAC(x, y, APDAKey_EL1, TRUE);
```

G1.1.3.53 AddPACDB

```
// AddPACDB()
// ==========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of x, y and the
// APDBKey_EL1.

bits(64) AddPACDB(bits(64) x, bits(64) y)
    constant bits(128) APDBKey_EL1 = APDBKeyHi_EL1<63:0> : APDBKeyLo_EL1<63:0>;
    if !IsAPDBKeyEnabled() then
        return x;
    else
        return AddPAC(x, y, APDBKey_EL1, TRUE);
```

G1.1.3.54 AddPACGA

```
// AddPACGA()
// ==========
// Returns a 64-bit value where the lower 32 bits are 0, and the upper 32 bits contain
// a 32-bit pointer authentication code which is derived using a cryptographic
// algorithm as a combination of x, y and the APGAKey_EL1.

bits(64) AddPACGA(bits(64) x, bits(64) y)
    boolean TrapEL2;
    constant bits(128) APGAKey_EL1 = APGAKeyHi_EL1<63:0> : APGAKeyLo_EL1<63:0>;

    case PSTATE.EL of
        when EL0
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
        when EL1
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
        when EL2
            TrapEL2 = FALSE;

    if TrapEL2 then
        TrapPACUse(EL2);
    else
        return ComputePAC(x, y, APGAKey_EL1<127:64>, APGAKey_EL1<63:0>)<63:32>:Zeros(32);
```

G1.1.3.55 AddPACIA

```
// AddPACIA()
// ==========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of x, y, and the
// APIAKey_EL1.

bits(64) AddPACIA(bits(64) x, bits(64) y)
    constant bits(128) APIAKey_EL1 = APIAKeyHi_EL1<63:0>:APIAKeyLo_EL1<63:0>;
    if !IsAPIAKeyEnabled() then
        return x;
    else
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        return AddPAC(x, y, APIAKey_EL1, FALSE);
```

G1.1.3.56 AddPACIB

```
// AddPACIB()
// ==========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of x, y and the
// APIBKey_EL1.

bits(64) AddPACIB(bits(64) x, bits(64) y)
    constant bits(128) APIBKey_EL1 = APIBKeyHi_EL1<63:0> : APIBKeyLo_EL1<63:0>;
    if !IsAPIBKeyEnabled() then
        return x;
    else
        return AddPAC(x, y, APIBKey_EL1, FALSE);
```

G1.1.3.57 AArch64.PACFailException

```
// AArch64.PACFailException()
// ==========================
// Generates a PAC Fail Exception

AArch64.PACFailException(bits(2) syndrome)
    route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';
    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception_PACFail);
    except.syndrome.iss<1:0>   = syndrome;
    except.syndrome.iss<24:2>  = Zeros(23);                // RES0

    if UInt(PSTATE.EL) > UInt(EL0) then
        AArch64.TakeException(PSTATE.EL, except, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```

G1.1.3.58 Auth

```
// Auth()
// ======
// Restores the upper bits of the address to be all zeros or all ones (based on the
// value of bit[55]) and computes and checks the pointer authentication code. If the
// check passes, then the restored address is returned. If the check fails, the
// second-top and third-top bits of the extension bits in the pointer authentication code
// field are corrupted to ensure that accessing the address will give a translation fault.

bits(64) Auth(bits(64) ptr, bits(64) modifier, bits(128) K, boolean data, bit key_number,
              boolean is_combined)
    bits(64) PAC;
    bits(64) result;
    bits(64) original_ptr;
    bits(2) error_code;
    bits(64) extfield;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    // Reconstruct the extension field used of adding the PAC to the pointer
    constant boolean tbi = EffectiveTBI(ptr, !data, PSTATE.EL) == '1';
    constant AddressSize bottom_PAC_bit = CalculateBottomPACBit(ptr<55>);
    extfield = Replicate(ptr<55>, 64);

    if tbi then
        if bottom_PAC_bit <= 55 then
            original_ptr = (ptr<63:56> :
                        extfield<55:bottom_PAC_bit> : ptr<bottom_PAC_bit-1:0>);
        else
            original_ptr = ptr<63:56> : ptr<55:0>;
    else
        original_ptr =  extfield<63:bottom_PAC_bit> : ptr<bottom_PAC_bit-1:0>;

    PAC = ComputePAC(original_ptr, modifier, K<127:64>, K<63:0>);
    // Check pointer authentication code
    if tbi then
        if !IsFeatureImplemented(FEAT_PAuth2) then
            assert (bottom_PAC_bit <= 52);
            if PAC<54:bottom_PAC_bit> == ptr<54:bottom_PAC_bit> then
                result = original_ptr;
            else
                error_code = key_number:NOT(key_number);
                result = original_ptr<63:55>:error_code:original_ptr<52:0>;
        else
            result = ptr;
            if (bottom_PAC_bit < 55) then
                result<54:bottom_PAC_bit> = result<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>;
            if (IsFeatureImplemented(FEAT_FPACCOMBINE) ||
                  (IsFeatureImplemented(FEAT_FPAC) && !is_combined)) then
                if (result<54:bottom_PAC_bit> !=
                      Replicate(result<55>, (55-bottom_PAC_bit))) then
                    error_code = (if data then '1' else '0'):key_number;
                    AArch64.PACFailException(error_code);
    else
        if !IsFeatureImplemented(FEAT_PAuth2) then
            assert (bottom_PAC_bit <= 52);
            if PAC<54:bottom_PAC_bit> == ptr<54:bottom_PAC_bit> && PAC<63:56> == ptr<63:56> then
                result = original_ptr;
            else
                error_code = key_number:NOT(key_number);
                result = original_ptr<63>:error_code:original_ptr<60:0>;
        else
            result = ptr;
            if bottom_PAC_bit < 55 then
                result<54:bottom_PAC_bit> = result<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>;
            result<63:56> = result<63:56> EOR PAC<63:56>;
            if (IsFeatureImplemented(FEAT_FPACCOMBINE) ||
                  (IsFeatureImplemented(FEAT_FPAC) && !is_combined)) then
                if (result<63:bottom_PAC_bit> !=
                      Replicate(result<55>, (64-bottom_PAC_bit))) then
                    error_code = (if data then '1' else '0'):key_number;
                    AArch64.PACFailException(error_code);
    return result;
```

G1.1.3.59 AuthDA

```
// AuthDA()
// ========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of x, using the same
// algorithm and key as AddPACDA().
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
bits(64) AuthDA(bits(64) x, bits(64) y, boolean is_combined)
    constant bits(128) APDAKey_EL1 = APDAKeyHi_EL1<63:0> : APDAKeyLo_EL1<63:0>;
    if !IsAPDAKeyEnabled() then
        return x;
    else
        return Auth(x, y, APDAKey_EL1, TRUE, '0', is_combined);
```

G1.1.3.60 AuthDB

```
// AuthDB()
// ========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a
// pointer authentication code in the pointer authentication code field bits of x, using
// the same algorithm and key as AddPACDB().

bits(64) AuthDB(bits(64) x, bits(64) y, boolean is_combined)
    constant bits(128) APDBKey_EL1 = APDBKeyHi_EL1<63:0> : APDBKeyLo_EL1<63:0>;
    if !IsAPDBKeyEnabled() then
        return x;
    else
        return Auth(x, y, APDBKey_EL1, TRUE, '1', is_combined);
```

G1.1.3.61 AuthIA

```
// AuthIA()
// ========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of x, using the same
// algorithm and key as AddPACIA().

bits(64) AuthIA(bits(64) x, bits(64) y, boolean is_combined)
    constant bits(128) APIAKey_EL1 = APIAKeyHi_EL1<63:0> : APIAKeyLo_EL1<63:0>;
    if !IsAPIAKeyEnabled() then
        return x;
    else
        return Auth(x, y, APIAKey_EL1, FALSE, '0', is_combined);
```

G1.1.3.62 AuthIB

```
// AuthIB()
// ========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of x, using the same
// algorithm and key as AddPACIB().

bits(64) AuthIB(bits(64) x, bits(64) y, boolean is_combined)
    constant bits(128) APIBKey_EL1 = APIBKeyHi_EL1<63:0> : APIBKeyLo_EL1<63:0>;
    if !IsAPIBKeyEnabled() then
        return x;
    else
        return Auth(x, y, APIBKey_EL1, FALSE, '1', is_combined);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.1.3.63 AArch64.PACEffectiveTxSZ

```
// AArch64.PACEffectiveTxSZ()
// ==========================
// Compute the effective value for TxSZ used to determine the placement of the PAC field

bits(6) AArch64.PACEffectiveTxSZ(Regime regime, S1TTWParams walkparams)
    constant integer s1maxtxsz = AArch64.MaxTxSZ(walkparams.tgx);
    constant integer s1mintxsz = AArch64.S1MinTxSZ(regime, walkparams);

    if AArch64.S1TxSZFaults(regime, walkparams) then
        if ConstrainUnpredictable(Unpredictable_RESTnSZ) == Constraint_FORCE then
            if UInt(walkparams.txsz) < s1mintxsz then
                return s1mintxsz<5:0>;
            if UInt(walkparams.txsz) > s1maxtxsz then
                return s1maxtxsz<5:0>;
    elsif UInt(walkparams.txsz) < s1mintxsz then
        return s1mintxsz<5:0>;
    elsif UInt(walkparams.txsz) > s1maxtxsz then
        return s1maxtxsz<5:0>;

    return walkparams.txsz;
```

G1.1.3.64 CalculateBottomPACBit

```
// CalculateBottomPACBit()
// =======================

AddressSize CalculateBottomPACBit(bit top_bit)
    Regime regime;
    S1TTWParams walkparams;
    AddressSize bottom_PAC_bit;

    regime = TranslationRegime(PSTATE.EL);
    if !AArch64.IsStage1VMSA(regime) then
        return AArch64.PAMax();

    ss = CurrentSecurityState();
    walkparams = AArch64.GetS1TTWParams(regime, PSTATE.EL, ss, Replicate(top_bit, 64));
    bottom_PAC_bit = 64 - UInt(AArch64.PACEffectiveTxSZ(regime, walkparams));

    return bottom_PAC_bit;
```

G1.1.3.65 ComputePAC

```
// ComputePAC()
// ============

bits(64) ComputePAC(bits(64) data, bits(64) modifier, bits(64) key0, bits(64) key1)
    if IsFeatureImplemented(FEAT_PACIMP) then
        return ComputePACIMPDEF(data, modifier, key0, key1);
    if IsFeatureImplemented(FEAT_PACQARMA3) then
        constant boolean isqarma3 = TRUE;
        return ComputePACQARMA(data, modifier, key0, key1, isqarma3);
    if IsFeatureImplemented(FEAT_PACQARMA5) then
        constant boolean isqarma3 = FALSE;
        return ComputePACQARMA(data, modifier, key0, key1, isqarma3);
    Unreachable();
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.1.3.66 ComputePAC2

```
// ComputePAC2()
// =============

bits(64) ComputePAC2(bits(64) data, bits(64) modifier1, bits(64) modifier2,
                     bits(64) key0, bits(64) key1)
    if IsFeatureImplemented(FEAT_PACIMP) then
        return ComputePAC2IMPDEF(data, modifier1, modifier2, key0, key1);
    if IsFeatureImplemented(FEAT_PACQARMA3) then
        constant boolean isqarma3 = TRUE;
        return ComputePAC2QARMA(data, modifier1, modifier2, key0, key1, isqarma3);
    if IsFeatureImplemented(FEAT_PACQARMA5) then
        constant boolean isqarma3 = FALSE;
        return ComputePAC2QARMA(data, modifier1, modifier2, key0, key1, isqarma3);
    Unreachable();
```

G1.1.3.67 ComputePAC2IMPDEF

```
// ComputePAC2IMPDEF()
// ===================
// Compute IMPLEMENTATION DEFINED cryptographic algorithm to be used for PAC calculation.

bits(64) ComputePAC2IMPDEF(bits(64) data, bits(64) modifier1, bits(64) modifier2, bits(64) key0,
                           bits(64) key1);
```

G1.1.3.68 ComputePAC2QARMA

```
// ComputePAC2QARMA()
// ==================

bits(64) ComputePAC2QARMA(bits(64) data, bits(64) modifier1, bits(64) modifier2, bits(64) key0,
                          bits(64) key1, boolean isqarma3)
    constant bits(64) concat_modifiers = modifier2<36:5>:modifier1<35:4>;
    return ComputePACQARMA(data, concat_modifiers, key0, key1, isqarma3);
```

G1.1.3.69 ComputePACIMPDEF

```
// ComputePACIMPDEF()
// ==================
// Compute IMPLEMENTATION DEFINED cryptographic algorithm to be used for PAC calculation.

bits(64) ComputePACIMPDEF(bits(64) data, bits(64) modifier, bits(64) key0, bits(64) key1);
```

G1.1.3.70 ComputePACQARMA

```
// ComputePACQARMA()
// =================
// Compute QARMA3 or QARMA5 cryptographic algorithm for PAC calculation

bits(64) ComputePACQARMA(bits(64) data, bits(64) modifier, bits(64) key0,
                         bits(64) key1, boolean isqarma3)
    bits(64)  workingval;
    bits(64)  runningmod;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
bits(64)  roundkey;
bits(64)  modk0;
constant bits(64) Alpha = 0xC0AC29B7C97C50DD<63:0>;

integer iterations;
RC[0] = 0x0000000000000000<63:0>;
RC[1] = 0x13198A2E03707344<63:0>;
RC[2] = 0xA4093822299F31D0<63:0>;

if isqarma3 then
    iterations = 2;
else // QARMA5
    iterations = 4;
    RC[3] = 0x082EFA98EC4E6C89<63:0>;
    RC[4] = 0x452821E638D01377<63:0>;

modk0 = key0<0>:key0<63:2>:(key0<63> EOR key0<1>);
runningmod = modifier;
workingval = data EOR key0;

for i = 0 to iterations
    roundkey = key1 EOR runningmod;
    workingval  = workingval EOR roundkey;
    workingval = workingval EOR RC[i];
    if i > 0 then
        workingval = PACCellShuffle(workingval);
        workingval = PACMult(workingval);
    if isqarma3 then
        workingval = PACSub1(workingval);
    else
        workingval = PACSub(workingval);
    runningmod = TweakShuffle(runningmod<63:0>);
roundkey = modk0 EOR runningmod;
workingval = workingval EOR roundkey;
workingval = PACCellShuffle(workingval);
workingval = PACMult(workingval);
if isqarma3 then
    workingval = PACSub1(workingval);
else
    workingval = PACSub(workingval);
workingval = PACCellShuffle(workingval);
workingval = PACMult(workingval);
workingval = key1 EOR workingval;
workingval = PACCellInvShuffle(workingval);
if isqarma3 then
    workingval = PACSub1(workingval);
else
    workingval = PACInvSub(workingval);
workingval = PACMult(workingval);
workingval = PACCellInvShuffle(workingval);
workingval = workingval EOR key0;
workingval = workingval EOR runningmod;
for i = 0 to iterations
    if isqarma3 then
        workingval = PACSub1(workingval);
    else
        workingval = PACInvSub(workingval);
    if i < iterations then
        workingval = PACMult(workingval);
        workingval = PACCellInvShuffle(workingval);
    runningmod = TweakInvShuffle(runningmod<63:0>);
    roundkey = key1 EOR runningmod;
    workingval = workingval EOR RC[iterations-i];
    workingval = workingval EOR roundkey;
    workingval = workingval EOR Alpha;
workingval = workingval EOR modk0;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    return workingval;
```

G1.1.3.71 PACCellInvShuffle

```
// PACCellInvShuffle()
// ===================

bits(64) PACCellInvShuffle(bits(64) indata)
    bits(64) outdata;
    outdata<3:0> = indata<15:12>;
    outdata<7:4> = indata<27:24>;
    outdata<11:8> = indata<51:48>;
    outdata<15:12> = indata<39:36>;
    outdata<19:16> = indata<59:56>;
    outdata<23:20> = indata<47:44>;
    outdata<27:24> = indata<7:4>;
    outdata<31:28> = indata<19:16>;
    outdata<35:32> = indata<35:32>;
    outdata<39:36> = indata<55:52>;
    outdata<43:40> = indata<31:28>;
    outdata<47:44> = indata<11:8>;
    outdata<51:48> = indata<23:20>;
    outdata<55:52> = indata<3:0>;
    outdata<59:56> = indata<43:40>;
    outdata<63:60> = indata<63:60>;
    return outdata;
```

G1.1.3.72 PACCellShuffle

```
// PACCellShuffle()
// ================

bits(64) PACCellShuffle(bits(64) indata)
    bits(64) outdata;
    outdata<3:0> = indata<55:52>;
    outdata<7:4> = indata<27:24>;
    outdata<11:8> = indata<47:44>;
    outdata<15:12> = indata<3:0>;
    outdata<19:16> = indata<31:28>;
    outdata<23:20> = indata<51:48>;
    outdata<27:24> = indata<7:4>;
    outdata<31:28> = indata<43:40>;
    outdata<35:32> = indata<35:32>;
    outdata<39:36> = indata<15:12>;
    outdata<43:40> = indata<59:56>;
    outdata<47:44> = indata<23:20>;
    outdata<51:48> = indata<11:8>;
    outdata<55:52> = indata<39:36>;
    outdata<59:56> = indata<19:16>;
    outdata<63:60> = indata<63:60>;
    return outdata;
```

G1.1.3.73 PACInvSub

```
// PACInvSub()
// ===========
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
bits(64) PACInvSub(bits(64) Tinput)
    // This is a 4-bit substitution from the PRINCE-family cipher
    bits(64) Toutput;
    for i = 0 to 15
        case Elem[Tinput, i, 4] of
            when '0000'  Elem[Toutput, i, 4] = '0101';
            when '0001'  Elem[Toutput, i, 4] = '1110';
            when '0010'  Elem[Toutput, i, 4] = '1101';
            when '0011'  Elem[Toutput, i, 4] = '1000';
            when '0100'  Elem[Toutput, i, 4] = '1010';
            when '0101'  Elem[Toutput, i, 4] = '1011';
            when '0110'  Elem[Toutput, i, 4] = '0001';
            when '0111'  Elem[Toutput, i, 4] = '1001';
            when '1000'  Elem[Toutput, i, 4] = '0010';
            when '1001'  Elem[Toutput, i, 4] = '0110';
            when '1010'  Elem[Toutput, i, 4] = '1111';
            when '1011'  Elem[Toutput, i, 4] = '0000';
            when '1100'  Elem[Toutput, i, 4] = '0100';
            when '1101'  Elem[Toutput, i, 4] = '1100';
            when '1110'  Elem[Toutput, i, 4] = '0111';
            when '1111'  Elem[Toutput, i, 4] = '0011';
    return Toutput;
```

G1.1.3.74 PACMult

```
// PACMult()
// =========

bits(64) PACMult(bits(64) Sinput)
    bits(4)  t0;
    bits(4)  t1;
    bits(4)  t2;
    bits(4)  t3;
    bits(64) Soutput;

    for i = 0 to 3
        t0<3:0> = ROL(Elem[Sinput, (i+8), 4], 1) EOR ROL(Elem[Sinput, (i+4), 4], 2);
        t0<3:0> = t0<3:0> EOR ROL(Elem[Sinput, i, 4], 1);
        t1<3:0> = ROL(Elem[Sinput, (i+12), 4], 1) EOR ROL(Elem[Sinput, (i+4), 4], 1);
        t1<3:0> = t1<3:0> EOR ROL(Elem[Sinput, i, 4], 2);
        t2<3:0> = ROL(Elem[Sinput, (i+12), 4], 2) EOR ROL(Elem[Sinput, (i+8), 4], 1);
        t2<3:0> = t2<3:0> EOR ROL(Elem[Sinput, i, 4], 1);
        t3<3:0> = ROL(Elem[Sinput, (i+12), 4], 1) EOR ROL(Elem[Sinput, (i+8), 4], 2);
        t3<3:0> = t3<3:0> EOR ROL(Elem[Sinput, (i+4), 4], 1);
        Elem[Soutput, i, 4] = t3<3:0>;
        Elem[Soutput, (i+4), 4] = t2<3:0>;
        Elem[Soutput, (i+8), 4] = t1<3:0>;
        Elem[Soutput, (i+12), 4] = t0<3:0>;
    return Soutput;
```

G1.1.3.75 PACSub

```
// PACSub()
// ========

bits(64) PACSub(bits(64) Tinput)
    // This is a 4-bit substitution from the PRINCE-family cipher
    bits(64) Toutput;
    for i = 0 to 15
        case Elem[Tinput, i, 4] of
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            when '0000'  Elem[Toutput, i, 4] = '1011';
            when '0001'  Elem[Toutput, i, 4] = '0110';
            when '0010'  Elem[Toutput, i, 4] = '1000';
            when '0011'  Elem[Toutput, i, 4] = '1111';
            when '0100'  Elem[Toutput, i, 4] = '1100';
            when '0101'  Elem[Toutput, i, 4] = '0000';
            when '0110'  Elem[Toutput, i, 4] = '1001';
            when '0111'  Elem[Toutput, i, 4] = '1110';
            when '1000'  Elem[Toutput, i, 4] = '0011';
            when '1001'  Elem[Toutput, i, 4] = '0111';
            when '1010'  Elem[Toutput, i, 4] = '0100';
            when '1011'  Elem[Toutput, i, 4] = '0101';
            when '1100'  Elem[Toutput, i, 4] = '1101';
            when '1101'  Elem[Toutput, i, 4] = '0010';
            when '1110'  Elem[Toutput, i, 4] = '0001';
            when '1111'  Elem[Toutput, i, 4] = '1010';
    return Toutput;
```

G1.1.3.76 PacSub1

```
// PacSub1()
// =========

bits(64) PACSub1(bits(64) Tinput)
    // This is a 4-bit substitution from Qarma sigma1
    bits(64) Toutput;
    for i = 0 to 15
        case Elem[Tinput, i, 4] of
            when '0000' Elem[Toutput, i, 4] = '1010';
            when '0001' Elem[Toutput, i, 4] = '1101';
            when '0010' Elem[Toutput, i, 4] = '1110';
            when '0011' Elem[Toutput, i, 4] = '0110';
            when '0100' Elem[Toutput, i, 4] = '1111';
            when '0101' Elem[Toutput, i, 4] = '0111';
            when '0110' Elem[Toutput, i, 4] = '0011';
            when '0111' Elem[Toutput, i, 4] = '0101';
            when '1000' Elem[Toutput, i, 4] = '1001';
            when '1001' Elem[Toutput, i, 4] = '1000';
            when '1010' Elem[Toutput, i, 4] = '0000';
            when '1011' Elem[Toutput, i, 4] = '1100';
            when '1100' Elem[Toutput, i, 4] = '1011';
            when '1101' Elem[Toutput, i, 4] = '0001';
            when '1110' Elem[Toutput, i, 4] = '0010';
            when '1111' Elem[Toutput, i, 4] = '0100';
    return Toutput;
```

G1.1.3.77 RC

```
// RC[]
// ====

array bits(64) RC[0..4];
```

G1.1.3.78 TweakCellInvRot

```
// TweakCellInvRot()
// =================
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
bits(4) TweakCellInvRot(bits(4) incell)
    bits(4) outcell;
    outcell<3> = incell<2>;
    outcell<2> = incell<1>;
    outcell<1> = incell<0>;
    outcell<0> = incell<0> EOR incell<3>;
    return outcell;
```

G1.1.3.79 TweakCellRot

```
// TweakCellRot()
// ==============

bits(4) TweakCellRot(bits(4) incell)
    bits(4) outcell;
    outcell<3> = incell<0> EOR incell<1>;
    outcell<2> = incell<3>;
    outcell<1> = incell<2>;
    outcell<0> = incell<1>;
    return outcell;
```

G1.1.3.80 TweakInvShuffle

```
// TweakInvShuffle()
// =================

bits(64) TweakInvShuffle(bits(64) indata)
    bits(64) outdata;
    outdata<3:0> = TweakCellInvRot(indata<51:48>);
    outdata<7:4> = indata<55:52>;
    outdata<11:8> = indata<23:20>;
    outdata<15:12> = indata<27:24>;
    outdata<19:16> = indata<3:0>;
    outdata<23:20> = indata<7:4>;
    outdata<27:24> = TweakCellInvRot(indata<11:8>);
    outdata<31:28> = indata<15:12>;
    outdata<35:32> = TweakCellInvRot(indata<31:28>);
    outdata<39:36> = TweakCellInvRot(indata<63:60>);
    outdata<43:40> = TweakCellInvRot(indata<59:56>);
    outdata<47:44> = TweakCellInvRot(indata<19:16>);
    outdata<51:48> = indata<35:32>;
    outdata<55:52> = indata<39:36>;
    outdata<59:56> = indata<43:40>;
    outdata<63:60> = TweakCellInvRot(indata<47:44>);
    return outdata;
```

G1.1.3.81 TweakShuffle

```
// TweakShuffle()
// ==============

bits(64) TweakShuffle(bits(64) indata)
    bits(64) outdata;
    outdata<3:0> = indata<19:16>;
    outdata<7:4> = indata<23:20>;
    outdata<11:8> = TweakCellRot(indata<27:24>);
    outdata<15:12> = indata<31:28>;
    outdata<19:16> = TweakCellRot(indata<47:44>);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    outdata<23:20> = indata<11:8>;
    outdata<27:24> = indata<15:12>;
    outdata<31:28> = TweakCellRot(indata<35:32>);
    outdata<35:32> = indata<51:48>;
    outdata<39:36> = indata<55:52>;
    outdata<43:40> = indata<59:56>;
    outdata<47:44> = TweakCellRot(indata<63:60>);
    outdata<51:48> = TweakCellRot(indata<3:0>);
    outdata<55:52> = indata<7:4>;
    outdata<59:56> = TweakCellRot(indata<43:40>);
    outdata<63:60> = TweakCellRot(indata<39:36>);
    return outdata;
```

G1.1.3.82 IsAPDAKeyEnabled

```
// IsAPDAKeyEnabled()
// ==================
// Returns TRUE if authentication using the APDAKey_EL1 key is enabled.
// Otherwise, depending on the state of the PE, generate a trap, or return FALSE.

boolean IsAPDAKeyEnabled()
    boolean TrapEL2;
    bits(1) Enable;

    case PSTATE.EL of
        when EL0
            Enable = SCTLR_EL1.EnDA;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnDA;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnDA;
            TrapEL2 = FALSE;

    if Enable == '0' then
        return FALSE;
    elsif TrapEL2 then
        TrapPACUse(EL2);
    else
        return TRUE;
```

G1.1.3.83 IsAPDBKeyEnabled

```
// IsAPDBKeyEnabled()
// ==================
// Returns TRUE if authentication using the APDBKey_EL1 key is enabled.
// Otherwise, depending on the state of the PE, generate a trap, or return FALSE.

boolean IsAPDBKeyEnabled()
    boolean TrapEL2;
    bits(1) Enable;

    case PSTATE.EL of
        when EL0
            Enable = SCTLR_EL1.EnDB;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnDB;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when EL2
            Enable = SCTLR_EL2.EnDB;
            TrapEL2 = FALSE;

    if Enable == '0' then
        return FALSE;
    elsif TrapEL2 then
        TrapPACUse(EL2);
    else
        return TRUE;
```

G1.1.3.84 IsAPIAKeyEnabled

```
// IsAPIAKeyEnabled()
// ==================
// Returns TRUE if authentication using the APIAKey_EL1 key is enabled.
// Otherwise, depending on the state of the PE, generate a trap, or return FALSE.

boolean IsAPIAKeyEnabled()
    boolean TrapEL2;
    bits(1) Enable;

    case PSTATE.EL of
        when EL0
            Enable = SCTLR_EL1.EnIA;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnIA;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnIA;
            TrapEL2 = FALSE;

    if Enable == '0' then
        return FALSE;
    elsif TrapEL2 then
        TrapPACUse(EL2);
    else
        return TRUE;
```

G1.1.3.85 IsAPIBKeyEnabled

```
// IsAPIBKeyEnabled()
// ==================
// Returns TRUE if authentication using the APIBKey_EL1 key is enabled.
// Otherwise, depending on the state of the PE, generate a trap, or return FALSE.

boolean IsAPIBKeyEnabled()
    boolean TrapEL2;
    bits(1) Enable;

    case PSTATE.EL of
        when EL0
            Enable = SCTLR_EL1.EnIB;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnIB;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnIB;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            TrapEL2 = FALSE;

    if Enable == '0' then
        return FALSE;
    elsif TrapEL2 then
        TrapPACUse(EL2);
    else
        return TRUE;
```

G1.1.3.86 Strip

```
// Strip()
// =======
// Strip() returns a 64-bit value containing A, but replacing the pointer authentication
// code field bits with the extension of the address bits. This can apply to either
// instructions or data, where, as the use of tagged pointers is distinct, it might be
// handled differently.

bits(64) Strip(bits(64) A, boolean data)
    bits(64) original_ptr;
    bits(64) extfield;
    constant boolean tbi = EffectiveTBI(A, !data, PSTATE.EL) == '1';
    constant AddressSize bottom_PAC_bit = CalculateBottomPACBit(A<55>);
    extfield = Replicate(A<55>, 64);

    if tbi then
        if (bottom_PAC_bit <= 55) then
            original_ptr = (A<63:56> :
                        extfield<55:bottom_PAC_bit> : A<bottom_PAC_bit-1:0>);
        else
            original_ptr = A<63:56> : A<55:0>;
    else
        original_ptr =  extfield<63:bottom_PAC_bit> : A<bottom_PAC_bit-1:0>;

    return original_ptr;
```

G1.1.3.87 TrapPACUse

```
// TrapPACUse()
// ============
// Used for the trapping of the pointer authentication functions by higher exception
// levels.

TrapPACUse(bits(2) target_el)
    assert HaveEL(target_el) && target_el !=  EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    ExceptionRecord except;
    vect_offset = 0;
    except = ExceptionSyndrome(Exception_PACTrap);
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

G1.1.3.88 AArch64.RestrictPrediction

```
// AArch64.RestrictPrediction()
// ============================
// Clear all predictions in the context.
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
AArch64.RestrictPrediction(bits(64) val, RestrictType restriction)

    ExecutionCntxt c;
    target_el    = val<25:24>;

    // If the target EL is not implemented or the instruction is executed at an
    // EL lower than the specified level, the instruction is treated as a NOP.
    if !HaveEL(target_el) || UInt(target_el) > UInt(PSTATE.EL) then ExecuteAsNOP();

    constant bit ns  = val<26>;
    ss = TargetSecurityState(ns);

    // If the combination of Security state and Exception level is not implemented,
    // the instruction is treated as a NOP.
    if target_el == EL3 && ss != SS_Secure then ExecuteAsNOP();

    c.security  = ss;
    c.target_el = target_el;

    if EL2Enabled() then
        if PSTATE.EL IN {EL0, EL1} then
            c.is_vmid_valid = TRUE;
            c.all_vmid      = FALSE;
            c.vmid          = VMID[];

        elsif target_el IN {EL0, EL1} then
            c.is_vmid_valid = TRUE;
            c.all_vmid      = val<48> == '1';
            c.vmid          = val<47:32>;       // Only valid if  val<48> == '0';

        else
            c.is_vmid_valid = FALSE;
    else
        c.is_vmid_valid = FALSE;

    if PSTATE.EL == EL0 then
        c.is_asid_valid = TRUE;
        c.all_asid      = FALSE;
        c.asid          = ASID[];

    elsif target_el == EL0 then
        c.is_asid_valid = TRUE;
        c.all_asid      = val<16> == '1';
        c.asid          = val<15:0>;            // Only valid if  val<16> == '0';

    else
        c.is_asid_valid = FALSE;

    c.restriction = restriction;
    RESTRICT_PREDICTIONS(c);
```

G1.1.3.89 Prefetch

```
// Prefetch()
// ==========

// Decode and execute the prefetch hint on ADDRESS specified by PRFOP

Prefetch(bits(64) address, bits(5) prfop)
    PrefetchHint hint;
    integer target;
    boolean stream;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    case prfop<4:3> of
        when '00' hint = Prefetch_READ;         // PLD: prefetch for load
        when '01' hint = Prefetch_EXEC;         // PLI: preload instructions
        when '10' hint = Prefetch_WRITE;        // PST: prepare for store
        when '11' return;                       // unallocated hint
    target = UInt(prfop<2:1>);                  // target cache level
    stream = (prfop<0> != '0');                 // streaming (non-temporal)
    Hint_Prefetch(address, hint, target, stream);
    return;
```

G1.1.3.90 PSTATEField

```
// PSTATEField
// ===========
// MSR (immediate) instruction destinations.

enumeration PSTATEField {PSTATEField_DAIFSet, PSTATEField_DAIFClr,
                         PSTATEField_PAN, // Armv8.1
                         PSTATEField_UAO, // Armv8.2
                         PSTATEField_DIT, // Armv8.4
                         PSTATEField_SSBS,
                         PSTATEField_SP
                         };
```

G1.1.3.91 AArch64.ESBOperation

```
// AArch64.ESBOperation()
// ======================
// Perform the AArch64 ESB operation, either for ESB executed in AArch64 state, or for
// ESB in AArch32 state when SError interrupts are routed to an Exception level using
// AArch64

AArch64.ESBOperation()
    bits(2) target_el;
    boolean masked;

    (masked, target_el) = PhysicalSErrorTarget();

    // Check for a masked Physical SError pending that can be synchronized
    // by an Error synchronization event.
    if masked && IsSynchronizablePhysicalSErrorPending() then
        constant boolean implicit_esb = FALSE;
        constant boolean is_esb = TRUE;
        bits(64) syndrome = Zeros(64);

        syndrome<31>   = '1'; // A
        syndrome<24:0> = AArch64.PhysicalSErrorSyndrome(is_esb, implicit_esb);
        DISR_EL1       = syndrome;
        ClearPendingPhysicalSError();               // Set ISR_EL1.A to 0

    return;
```

G1.1.3.92 AArch64.EncodeAsyncErrorSyndrome

```
// AArch64.EncodeAsyncErrorSyndrome()
// ==================================
// Return the encoding for specified ErrorState for an SError exception taken
// to AArch64 state.
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
bits(3) AArch64.EncodeAsyncErrorSyndrome(ErrorState errorstate)
    case errorstate of
        when ErrorState_UC  return '000';
        when ErrorState_UEU return '001';
        when ErrorState_UEO return '010';
        when ErrorState_UER return '011';
        when ErrorState_CE  return '110';
        otherwise Unreachable();
```

G1.1.3.93 AArch64.EncodeSyncErrorSyndrome

```
// AArch64.EncodeSyncErrorSyndrome()
// =================================
// Return the encoding for specified ErrorState for a synchronous Abort
// exception taken to AArch64 state.

bits(2) AArch64.EncodeSyncErrorSyndrome(ErrorState errorstate)
    case errorstate of
        when ErrorState_UC  return '10';
        when ErrorState_UEU return '10';    // UEU is reported as UC
        when ErrorState_UEO return '11';
        when ErrorState_UER return '00';
        otherwise Unreachable();
```

G1.1.3.94 AArch64.PhysicalSErrorSyndrome

```
// AArch64.PhysicalSErrorSyndrome()
// ================================
// Generate SError syndrome.

bits(25) AArch64.PhysicalSErrorSyndrome(boolean is_esb, boolean implicit_esb)
    bits(25) syndrome = Zeros(25);

    if ReportErrorAsUncategorized() then
        syndrome = Zeros(25);
    elsif ReportErrorAsIMPDEF() then
        syndrome<24>   = '1';                                          // IDS
        syndrome<23:0> = bits(24) IMPLEMENTATION_DEFINED "IMPDEF ErrorState";
    else
        constant FaultRecord fault = GetPendingPhysicalSError();
        constant ErrorState errorstate = PEErrorState(fault);
        syndrome<24>    = '0';                                         // IDS
        syndrome<13>    = if implicit_esb then '1' else '0';           // IESB
        syndrome<12:10> = AArch64.EncodeAsyncErrorSyndrome(errorstate);// AET
        syndrome<9>     = fault.extflag;                               // EA
        syndrome<5:0>   = '010001';                                    // DFSC

    return syndrome;
```

G1.1.3.95 AArch64.vESBOperation

```
// AArch64.vESBOperation()
// =======================
// Perform the AArch64 ESB operation for an unmasked pending virtual SError exception.

AArch64.vESBOperation()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !ELUsingAArch32(EL2);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    // If physical SError exceptions are routed to EL2, and TGE is not set, then a virtual
    // SError exception might be pending.
    vsei_pending = (IsVirtualSErrorPending() && EffectiveTGE() == '0' &&
                    EffectiveHCR_AMO() == '1');
    vsei_masked = PSTATE.A == '1' || Halted() || ExternalDebugInterruptsDisabled(EL1);

    // Check for a masked virtual SError pending
    if vsei_pending && vsei_masked then
        bits(64) target = Zeros(64);
        target<31> = '1';  // A
        target<24:0> = VSESR_EL2<24:0>;
        VDISR_EL2 = target;
        ClearPendingVirtualSError();

    return;
```

G1.1.3.96 FirstRecordOfNode

```
// FirstRecordOfNode()
// ===================
// Return the first record in the node that contains the record n.

integer FirstRecordOfNode(integer n)
    for q = n downto 0
        if IsFirstRecordOfNode(q) then return q;
    Unreachable();
```

G1.1.3.97 IsCommonFaultInjectionImplemented

```
// IsCommonFaultInjectionImplemented()
// ===================================
// Check if the Common Fault Injection Model Extension is implemented by the node that owns this
// error record.

boolean IsCommonFaultInjectionImplemented(integer n);
```

G1.1.3.98 IsCountableErrorsRecorded

```
// IsCountableErrorsRecorded()
// ===========================
// Check whether Error record n records countable errors.

boolean IsCountableErrorsRecorded(integer n);
```

G1.1.3.99 IsErrorAddressIncluded

```
// IsErrorAddressIncluded()
// ========================
// Check whether Error record n includes an address associated with an error.

boolean IsErrorAddressIncluded(integer n);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.1.3.100 IsErrorRecordImplemented

```
// IsErrorRecordImplemented()
// ==========================
// Is the error record n implemented

boolean IsErrorRecordImplemented(integer n);
```

G1.1.3.101 IsFirstRecordOfNode

```
// IsFirstRecordOfNode()
// =====================
// Check if the record q is the first error record in its node.

boolean IsFirstRecordOfNode(integer q);
```

G1.1.3.102 IsSPMUCounterImplemented

```
// IsSPMUCounterImplemented()
// ==========================
// Does the System PMU s implement the counter n.

boolean IsSPMUCounterImplemented(integer s, integer n);
```

G1.1.3.103 FPReduce

```
// FPReduce()
// ==========
// Perform the floating-point operation 'op' on pairs of elements from the input vector,
// reducing the vector to a scalar result.

bits(esize) FPReduce(ReduceOp op, bits(N) input, integer esize, FPCR_Type fpcr)
    bits(esize) hi;
    bits(esize) lo;
    bits(esize) result;
    constant integer half = N DIV 2;

    if N == esize then
        return input<esize-1:0>;

    hi = FPReduce(op, input<N-1:half>, esize, fpcr);
    lo = FPReduce(op, input<half-1:0>, esize, fpcr);
    case op of
        when ReduceOp_FMINNUM
            result = FPMinNum(lo, hi, fpcr);
        when ReduceOp_FMAXNUM
            result = FPMaxNum(lo, hi, fpcr);
        when ReduceOp_FMIN
            result = FPMin(lo, hi, fpcr);
        when ReduceOp_FMAX
            result = FPMax(lo, hi, fpcr);
        when ReduceOp_FADD
            result = FPAdd(lo, hi, fpcr);

    return result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.1.3.104 IntReduce

```
// IntReduce()
// ===========
// Perform the integer operation 'op' on pairs of elements from the input vector,
// reducing the vector to a scalar result.

bits(esize) IntReduce(ReduceOp op, bits(N) input, integer esize)
    bits(esize) hi;
    bits(esize) lo;
    bits(esize) result;
    constant integer half = N DIV 2;
    if N == esize then
        return input<esize-1:0>;

    hi = IntReduce(op, input<N-1:half>, esize);
    lo = IntReduce(op, input<half-1:0>, esize);
    case op of
        when ReduceOp_ADD
            result = lo + hi;

    return result;
```

G1.1.3.105 ReduceOp

```
// ReduceOp
// ========
// Vector reduce instruction types.

enumeration ReduceOp {ReduceOp_FMINNUM, ReduceOp_FMAXNUM,
                      ReduceOp_FMIN, ReduceOp_FMAX,
                      ReduceOp_FADD, ReduceOp_ADD};
```

G1.1.3.106 AArch64.MaybeZeroRegisterUppers

```
// AArch64.MaybeZeroRegisterUppers()
// =================================
// On taking an exception to  AArch64 from AArch32, it is CONSTRAINED UNPREDICTABLE whether the top
// 32 bits of registers visible at any lower Exception level using AArch32 are set to zero.

AArch64.MaybeZeroRegisterUppers()
    assert UsingAArch32();         // Always called from AArch32 state before entering AArch64 state

    integer first;
    integer last;
    boolean include_R15;
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then
        first = 0;  last = 14;  include_R15 = FALSE;
    elsif PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !ELUsingAArch32(EL2) then
        first = 0;  last = 30;  include_R15 = FALSE;
    else
        first = 0;  last = 30;  include_R15 = TRUE;

    for n = first to last
        if (n != 15 || include_R15) && ConstrainUnpredictableBool(Unpredictable_ZEROUPPER) then
            _R[n]<63:32> = Zeros(32);

    return;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.1.3.107 AArch64.ResetGeneralRegisters

```
// AArch64.ResetGeneralRegisters()
// ===============================

AArch64.ResetGeneralRegisters()

    for i = 0 to 30
        X[i, 64] = bits(64) UNKNOWN;

    return;
```

G1.1.3.108 AArch64.ResetSIMDFPRegisters

```
// AArch64.ResetSIMDFPRegisters()
// ==============================

AArch64.ResetSIMDFPRegisters()

    for i = 0 to 31
        V[i, 128] = bits(128) UNKNOWN;

    return;
```

G1.1.3.109 AArch64.ResetSpecialRegisters

```
// AArch64.ResetSpecialRegisters()
// ===============================

AArch64.ResetSpecialRegisters()

    // AArch64 special registers
    SP_EL0 = bits(64) UNKNOWN;
    SP_EL1 = bits(64) UNKNOWN;

    SPSR_EL1 = bits(64) UNKNOWN;
    ELR_EL1  = bits(64) UNKNOWN;
    if HaveEL(EL2) then
        SP_EL2 = bits(64) UNKNOWN;
        SPSR_EL2 = bits(64) UNKNOWN;
        ELR_EL2  = bits(64) UNKNOWN;

    // External debug special registers
    DLR_EL0 = bits(64) UNKNOWN;
    DSPSR_EL0 = bits(64) UNKNOWN;

    return;
```

G1.1.3.110 AArch64.ResetSystemRegisters

```
// AArch64.ResetSystemRegisters()
// ==============================

AArch64.ResetSystemRegisters(boolean cold_reset);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.1.3.111 PC64

```
// Program counter
// ++++++++++++++++

// PC64 - getter
// =============
// Read 64-bit program counter.

bits(64) PC64
    return _PC;
```

G1.1.3.112 SP

```
// SP - setter
// ===========
// Write a 32-bit or 64-bit value to the current stack pointer.

SP[integer width] = bits(width) value
    assert width IN {64, 32};
    if PSTATE.SP == '0' then
        SP_EL0 = ZeroExtend(value, 64);
    else
        case PSTATE.EL of
            when EL0  SP_EL0 = ZeroExtend(value, 64);
            when EL1  SP_EL1 = ZeroExtend(value, 64);
            when EL2  SP_EL2 = ZeroExtend(value, 64);
            when EL3  Unreachable();
    return;

// SP - getter
// ===========
// Read the least-significant 32 or 64 bits from the current stack pointer.

bits(width) SP[integer width]
    assert width IN {64, 32};
    if PSTATE.SP == '0' then
        return SP_EL0<width-1:0>;
    else
        case PSTATE.EL of
            when EL0  return SP_EL0<width-1:0>;
            when EL1  return SP_EL1<width-1:0>;
            when EL2  return SP_EL2<width-1:0>;
            when EL3 Unreachable();
```

G1.1.3.113 SPMSCR_EL1

```
// SPMSCR_EL1 - setter
// ===================
// Configure the observability of secure events by System Performance
// Monitor for System PMU 's'.

SPMSCR_EL1[integer s] = bits(64) value;
```

G1.1.3.114 X

```
// X - setter
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// ==========
// Write a 32-bit or 64-bit value to a general-purpose register.

X[integer n, integer width] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width IN {32,64};
    if n != 31 then
        _R[n] = ZeroExtend(value, 64);
    return;

// X - getter
// ==========
// Read the least-significant 8, 16, 32, or 64 bits from a general-purpose register.

bits(width) X[integer n, integer width]
    assert n >= 0 && n <= 31;
    assert width IN {8, 16, 32, 64};
    constant rw = width;
    if n != 31 then
        return _R[n]<rw-1:0>;
    else
        return Zeros(rw);

// X - setter
// ==========
// Write a 128-bit value to two general-purpose registers.

X[integer lr, integer hr, integer width] = bits(width) value
    assert width == 128;
    constant integer half = width DIV 2;
    X[lr, half] = value<0+:half>;
    X[hr, half] = value<half+:half>;
    return;

// X - getter
// ==========
// Read 128 bits from two separate general-purpose registers.

bits(width) X[integer lr, integer hr, integer width]
    assert width == 128;
    constant integer half = width DIV 2;
    return X[hr, half] : X[lr, half];
```

G1.1.3.115 DecodeShift

```
// DecodeShift()
// =============
// Decode shift encodings

ShiftType DecodeShift(bits(2) op)
    case op of
        when '00'  return ShiftType_LSL;
        when '01'  return ShiftType_LSR;
        when '10'  return ShiftType_ASR;
        when '11'  return ShiftType_ROR;
```

G1.1.3.116 ShiftReg

```
// ShiftReg()
// ==========
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// Perform shift of a register operand

bits(N) ShiftReg(integer reg, ShiftType shiftype, integer amount, integer N)
    bits(N) result = X[reg, N];
    case shiftype of
        when ShiftType_LSL result = LSL(result, amount);
        when ShiftType_LSR result = LSR(result, amount);
        when ShiftType_ASR result = ASR(result, amount);
        when ShiftType_ROR result = ROR(result, amount);
    return result;
```

G1.1.3.117 ShiftType

```
// ShiftType
// =========
// AArch64 register shifts.

enumeration ShiftType   {ShiftType_LSL, ShiftType_LSR, ShiftType_ASR, ShiftType_ROR};
```

G1.1.3.118 SystemHintOp

```
// SystemHintOp
// ============
// System Hint instruction types.

enumeration SystemHintOp {
    SystemHintOp_NOP,
    SystemHintOp_YIELD,
    SystemHintOp_WFE,
    SystemHintOp_WFI,
    SystemHintOp_SEV,
    SystemHintOp_SEVL,
    SystemHintOp_DGH,
    SystemHintOp_ESB,
    SystemHintOp_TSB,
    SystemHintOp_CSDB
};
```

G1.1.3.119 SysLOp

```
// SysLOp()
// ========

SystemLOp SysLOp(bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2)
    case op1:CRn:CRm:op2 of
        otherwise               return Sysl_SYSL;
```

G1.1.3.120 SystemLOp

```
// SystemLOp
// =========
// System instruction with result types.

enumeration SystemLOp {
    Sysl_SYSL
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
};
```

G1.1.3.121 SysOp

```
// SysOp()
// =======

SystemOp SysOp(bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2)
    case op1:CRn:CRm:op2 of
        when '000 0111 1000 000' return Sys_AT;    // S1E1R
        when '000 0111 1000 001' return Sys_AT;    // S1E1W
        when '000 0111 1000 010' return Sys_AT;    // S1E0R
        when '000 0111 1000 011' return Sys_AT;    // S1E0W
        when '000 0111 1001 000' return Sys_AT;    // S1E1RP
        when '000 0111 1001 001' return Sys_AT;    // S1E1WP
        when '100 0111 1000 000' return Sys_AT;    // S1E2R
        when '100 0111 1000 001' return Sys_AT;    // S1E2W
        when '100 0111 1000 100' return Sys_AT;    // S12E1R
        when '100 0111 1000 101' return Sys_AT;    // S12E1W
        when '100 0111 1000 110' return Sys_AT;    // S12E0R
        when '100 0111 1000 111' return Sys_AT;    // S12E0W
        when '110 0111 1000 000' return Sys_AT;    // S1E3R
        when '110 0111 1000 001' return Sys_AT;    // S1E3W
        when '000 0111 0110 001' return Sys_DC;    // IVAC
        when '000 0111 0110 010' return Sys_DC;    // ISW
        when '000 0111 1010 010' return Sys_DC;    // CSW
        when '000 0111 1110 010' return Sys_DC;    // CISW
        when '011 0111 0100 001' return Sys_DC;    // ZVA
        when '011 0111 1010 001' return Sys_DC;    // CVAC
        when '011 0111 1011 001' return Sys_DC;    // CVAU
        when '011 0111 1100 001' return Sys_DC;    // CVAP
        when '011 0111 1101 001' return Sys_DC;    // CVADP
        when '011 0111 1110 001' return Sys_DC;    // CIVAC
        when '000 0111 0001 000' return Sys_IC;    // IALLUIS
        when '000 0111 0101 000' return Sys_IC;    // IALLU
        when '011 0111 0101 001' return Sys_IC;    // IVAU
        when '000 1000 0001 000' return Sys_TLBI;  // VMALLE1OS
        when '000 1000 0001 001' return Sys_TLBI;  // VAE1OS
        when '000 1000 0001 010' return Sys_TLBI;  // ASIDE1OS
        when '000 1000 0001 011' return Sys_TLBI;  // VAAE1OS
        when '000 1000 0001 101' return Sys_TLBI;  // VALE1OS
        when '000 1000 0001 111' return Sys_TLBI;  // VAALE1OS
        when '000 1000 0010 001' return Sys_TLBI;  // RVAE1IS
        when '000 1000 0010 011' return Sys_TLBI;  // RVAAE1IS
        when '000 1000 0010 101' return Sys_TLBI;  // RVALE1IS
        when '000 1000 0010 111' return Sys_TLBI;  // RVAALE1IS
        when '000 1000 0011 000' return Sys_TLBI;  // VMALLE1IS
        when '000 1000 0011 001' return Sys_TLBI;  // VAE1IS
        when '000 1000 0011 010' return Sys_TLBI;  // ASIDE1IS
        when '000 1000 0011 011' return Sys_TLBI;  // VAAE1IS
        when '000 1000 0011 101' return Sys_TLBI;  // VALE1IS
        when '000 1000 0011 111' return Sys_TLBI;  // VAALE1IS
        when '000 1000 0101 001' return Sys_TLBI;  // RVAE1OS
        when '000 1000 0101 011' return Sys_TLBI;  // RVAAE1OS
        when '000 1000 0101 101' return Sys_TLBI;  // RVALE1OS
        when '000 1000 0101 111' return Sys_TLBI;  // RVAALE1OS
        when '000 1000 0110 001' return Sys_TLBI;  // RVAE1
        when '000 1000 0110 011' return Sys_TLBI;  // RVAAE1
        when '000 1000 0110 101' return Sys_TLBI;  // RVALE1
        when '000 1000 0110 111' return Sys_TLBI;  // RVAALE1
        when '000 1000 0111 000' return Sys_TLBI;  // VMALLE1
        when '000 1000 0111 001' return Sys_TLBI;  // VAE1
        when '000 1000 0111 010' return Sys_TLBI;  // ASIDE1
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when '000 1000 0111 011' return Sys_TLBI;  // VAAE1
        when '000 1000 0111 101' return Sys_TLBI;  // VALE1
        when '000 1000 0111 111' return Sys_TLBI;  // VAALE1
        when '100 1000 0000 001' return Sys_TLBI;  // IPAS2E1IS
        when '100 1000 0000 010' return Sys_TLBI;  // RIPAS2E1IS
        when '100 1000 0000 101' return Sys_TLBI;  // IPAS2LE1IS
        when '100 1000 0000 110' return Sys_TLBI;  // RIPAS2LE1IS
        when '100 1000 0001 000' return Sys_TLBI;  // ALLE2OS
        when '100 1000 0001 001' return Sys_TLBI;  // VAE2OS
        when '100 1000 0001 100' return Sys_TLBI;  // ALLE1OS
        when '100 1000 0001 101' return Sys_TLBI;  // VALE2OS
        when '100 1000 0001 110' return Sys_TLBI;  // VMALLS12E1OS
        when '100 1000 0010 001' return Sys_TLBI;  // RVAE2IS
        when '100 1000 0010 101' return Sys_TLBI;  // RVALE2IS
        when '100 1000 0011 000' return Sys_TLBI;  // ALLE2IS
        when '100 1000 0011 001' return Sys_TLBI;  // VAE2IS
        when '100 1000 0011 100' return Sys_TLBI;  // ALLE1IS
        when '100 1000 0011 101' return Sys_TLBI;  // VALE2IS
        when '100 1000 0011 110' return Sys_TLBI;  // VMALLS12E1IS
        when '100 1000 0100 000' return Sys_TLBI;  // IPAS2E1OS
        when '100 1000 0100 001' return Sys_TLBI;  // IPAS2E1
        when '100 1000 0100 010' return Sys_TLBI;  // RIPAS2E1
        when '100 1000 0100 011' return Sys_TLBI;  // RIPAS2E1OS
        when '100 1000 0100 100' return Sys_TLBI;  // IPAS2LE1OS
        when '100 1000 0100 101' return Sys_TLBI;  // IPAS2LE1
        when '100 1000 0100 110' return Sys_TLBI;  // RIPAS2LE1
        when '100 1000 0100 111' return Sys_TLBI;  // RIPAS2LE1OS
        when '100 1000 0101 001' return Sys_TLBI;  // RVAE2OS
        when '100 1000 0101 101' return Sys_TLBI;  // RVALE2OS
        when '100 1000 0110 001' return Sys_TLBI;  // RVAE2
        when '100 1000 0110 101' return Sys_TLBI;  // RVALE2
        when '100 1000 0111 000' return Sys_TLBI;  // ALLE2
        when '100 1000 0111 001' return Sys_TLBI;  // VAE2
        when '100 1000 0111 100' return Sys_TLBI;  // ALLE1
        when '100 1000 0111 101' return Sys_TLBI;  // VALE2
        when '100 1000 0111 110' return Sys_TLBI;  // VMALLS12E1
        when '110 1000 0001 000' return Sys_TLBI;  // ALLE3OS
        when '110 1000 0001 001' return Sys_TLBI;  // VAE3OS
        when '110 1000 0001 101' return Sys_TLBI;  // VALE3OS
        when '110 1000 0010 001' return Sys_TLBI;  // RVAE3IS
        when '110 1000 0010 101' return Sys_TLBI;  // RVALE3IS
        when '110 1000 0011 000' return Sys_TLBI;  // ALLE3IS
        when '110 1000 0011 001' return Sys_TLBI;  // VAE3IS
        when '110 1000 0011 101' return Sys_TLBI;  // VALE3IS
        when '110 1000 0101 001' return Sys_TLBI;  // RVAE3OS
        when '110 1000 0101 101' return Sys_TLBI;  // RVALE3OS
        when '110 1000 0110 001' return Sys_TLBI;  // RVAE3
        when '110 1000 0110 101' return Sys_TLBI;  // RVALE3
        when '110 1000 0111 000' return Sys_TLBI;  // ALLE3
        when '110 1000 0111 001' return Sys_TLBI;  // VAE3
        when '110 1000 0111 101' return Sys_TLBI;  // VALE3
        otherwise                return Sys_SYS;
```

G1.1.3.122 SystemOp

```
// SystemOp
// ========
// System instruction types.

enumeration SystemOp {
    Sys_AT,
    Sys_DC,
    Sys_IC,
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    Sys_TLBI,
    Sys_SYS
};
```

G1.1.3.123 SysOp128

```
// SysOp128()
// ==========

SystemOp128 SysOp128(bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2)
    case op1:CRn:CRm:op2 of
        when '000 1000 0001 001' return Sys_TLBIP;  // VAE1OS
        when '000 1000 0001 011' return Sys_TLBIP;  // VAAE1OS
        when '000 1000 0001 101' return Sys_TLBIP;  // VALE1OS
        when '000 1000 0001 111' return Sys_TLBIP;  // VAALE1OS
        when '000 1000 0011 001' return Sys_TLBIP;  // VAE1IS
        when '000 1000 0011 011' return Sys_TLBIP;  // VAAE1IS
        when '000 1000 0011 101' return Sys_TLBIP;  // VALE1IS
        when '000 1000 0011 111' return Sys_TLBIP;  // VAALE1IS
        when '000 1000 0111 001' return Sys_TLBIP;  // VAE1
        when '000 1000 0111 011' return Sys_TLBIP;  // VAAE1
        when '000 1000 0111 101' return Sys_TLBIP;  // VALE1
        when '000 1000 0111 111' return Sys_TLBIP;  // VAALE1
        when '000 1001 0001 001' return Sys_TLBIP;  // VAE1OSNXS
        when '000 1001 0001 011' return Sys_TLBIP;  // VAAE1OSNXS
        when '000 1001 0001 101' return Sys_TLBIP;  // VALE1OSNXS
        when '000 1001 0001 111' return Sys_TLBIP;  // VAALE1OSNXS
        when '000 1001 0011 001' return Sys_TLBIP;  // VAE1ISNXS
        when '000 1001 0011 011' return Sys_TLBIP;  // VAAE1ISNXS
        when '000 1001 0011 101' return Sys_TLBIP;  // VALE1ISNXS
        when '000 1001 0011 111' return Sys_TLBIP;  // VAALE1ISNXS
        when '000 1001 0111 001' return Sys_TLBIP;  // VAE1NXS
        when '000 1001 0111 011' return Sys_TLBIP;  // VAAE1NXS
        when '000 1001 0111 101' return Sys_TLBIP;  // VALE1NXS
        when '000 1001 0111 111' return Sys_TLBIP;  // VAALE1NXS
        when '100 1000 0001 001' return Sys_TLBIP;  // VAE2OS
        when '100 1000 0001 101' return Sys_TLBIP;  // VALE2OS
        when '100 1000 0011 001' return Sys_TLBIP;  // VAE2IS
        when '100 1000 0011 101' return Sys_TLBIP;  // VALE2IS
        when '100 1000 0111 001' return Sys_TLBIP;  // VAE2
        when '100 1000 0111 101' return Sys_TLBIP;  // VALE2
        when '100 1001 0001 001' return Sys_TLBIP;  // VAE2OSNXS
        when '100 1001 0001 101' return Sys_TLBIP;  // VALE2OSNXS
        when '100 1001 0011 001' return Sys_TLBIP;  // VAE2ISNXS
        when '100 1001 0011 101' return Sys_TLBIP;  // VALE2ISNXS
        when '100 1001 0111 001' return Sys_TLBIP;  // VAE2NXS
        when '100 1001 0111 101' return Sys_TLBIP;  // VALE2NXS
        when '110 1000 0001 001' return Sys_TLBIP;  // VAE3OS
        when '110 1000 0001 101' return Sys_TLBIP;  // VALE3OS
        when '110 1000 0011 001' return Sys_TLBIP;  // VAE3IS
        when '110 1000 0011 101' return Sys_TLBIP;  // VALE3IS
        when '110 1000 0111 001' return Sys_TLBIP;  // VAE3
        when '110 1000 0111 101' return Sys_TLBIP;  // VALE3
        when '110 1001 0001 001' return Sys_TLBIP;  // VAE3OSNXS
        when '110 1001 0001 101' return Sys_TLBIP;  // VALE3OSNXS
        when '110 1001 0011 001' return Sys_TLBIP;  // VAE3ISNXS
        when '110 1001 0011 101' return Sys_TLBIP;  // VALE3ISNXS
        when '110 1001 0111 001' return Sys_TLBIP;  // VAE3NXS
        when '110 1001 0111 101' return Sys_TLBIP;  // VALE3NXS
        when '100 1000 0000 001' return Sys_TLBIP;  // IPAS2E1IS
        when '100 1000 0000 101' return Sys_TLBIP;  // IPAS2LE1IS
        when '100 1000 0100 000' return Sys_TLBIP;  // IPAS2E1OS
        when '100 1000 0100 001' return Sys_TLBIP;  // IPAS2E1
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
when '100 1000 0100 100' return Sys_TLBIP;  // IPAS2LE1OS
when '100 1000 0100 101' return Sys_TLBIP;  // IPAS2LE1
when '100 1001 0000 001' return Sys_TLBIP;  // IPAS2E1ISNXS
when '100 1001 0000 101' return Sys_TLBIP;  // IPAS2LE1ISNXS
when '100 1001 0100 000' return Sys_TLBIP;  // IPAS2E1OSNXS
when '100 1001 0100 001' return Sys_TLBIP;  // IPAS2E1NXS
when '100 1001 0100 100' return Sys_TLBIP;  // IPAS2LE1OSNXS
when '100 1001 0100 101' return Sys_TLBIP;  // IPAS2LE1NXS
when '000 1000 0010 001' return Sys_TLBIP;  // RVAE1IS
when '000 1000 0010 011' return Sys_TLBIP;  // RVAAE1IS
when '000 1000 0010 101' return Sys_TLBIP;  // RVALE1IS
when '000 1000 0010 111' return Sys_TLBIP;  // RVAALE1IS
when '000 1000 0101 001' return Sys_TLBIP;  // RVAE1OS
when '000 1000 0101 011' return Sys_TLBIP;  // RVAAE1OS
when '000 1000 0101 101' return Sys_TLBIP;  // RVALE1OS
when '000 1000 0101 111' return Sys_TLBIP;  // RVAALE1OS
when '000 1000 0110 001' return Sys_TLBIP;  // RVAE1
when '000 1000 0110 011' return Sys_TLBIP;  // RVAAE1
when '000 1000 0110 101' return Sys_TLBIP;  // RVALE1
when '000 1000 0110 111' return Sys_TLBIP;  // RVAALE1
when '000 1001 0010 001' return Sys_TLBIP;  // RVAE1ISNXS
when '000 1001 0010 011' return Sys_TLBIP;  // RVAAE1ISNXS
when '000 1001 0010 101' return Sys_TLBIP;  // RVALE1ISNXS
when '000 1001 0010 111' return Sys_TLBIP;  // RVAALE1ISNXS
when '000 1001 0101 001' return Sys_TLBIP;  // RVAE1OSNXS
when '000 1001 0101 011' return Sys_TLBIP;  // RVAAE1OSNXS
when '000 1001 0101 101' return Sys_TLBIP;  // RVALE1OSNXS
when '000 1001 0101 111' return Sys_TLBIP;  // RVAALE1OSNXS
when '000 1001 0110 001' return Sys_TLBIP;  // RVAE1NXS
when '000 1001 0110 011' return Sys_TLBIP;  // RVAAE1NXS
when '000 1001 0110 101' return Sys_TLBIP;  // RVALE1NXS
when '000 1001 0110 111' return Sys_TLBIP;  // RVAALE1NXS
when '100 1000 0010 001' return Sys_TLBIP;  // RVAE2IS
when '100 1000 0010 101' return Sys_TLBIP;  // RVALE2IS
when '100 1000 0101 001' return Sys_TLBIP;  // RVAE2OS
when '100 1000 0101 101' return Sys_TLBIP;  // RVALE2OS
when '100 1000 0110 001' return Sys_TLBIP;  // RVAE2
when '100 1000 0110 101' return Sys_TLBIP;  // RVALE2
when '100 1001 0010 001' return Sys_TLBIP;  // RVAE2ISNXS
when '100 1001 0010 101' return Sys_TLBIP;  // RVALE2ISNXS
when '100 1001 0101 001' return Sys_TLBIP;  // RVAE2OSNXS
when '100 1001 0101 101' return Sys_TLBIP;  // RVALE2OSNXS
when '100 1001 0110 001' return Sys_TLBIP;  // RVAE2NXS
when '100 1001 0110 101' return Sys_TLBIP;  // RVALE2NXS
when '110 1000 0010 001' return Sys_TLBIP;  // RVAE3IS
when '110 1000 0010 101' return Sys_TLBIP;  // RVALE3IS
when '110 1000 0101 001' return Sys_TLBIP;  // RVAE3OS
when '110 1000 0101 101' return Sys_TLBIP;  // RVALE3OS
when '110 1000 0110 001' return Sys_TLBIP;  // RVAE3
when '110 1000 0110 101' return Sys_TLBIP;  // RVALE3
when '110 1001 0010 001' return Sys_TLBIP;  // RVAE3ISNXS
when '110 1001 0010 101' return Sys_TLBIP;  // RVALE3ISNXS
when '110 1001 0101 001' return Sys_TLBIP;  // RVAE3OSNXS
when '110 1001 0101 101' return Sys_TLBIP;  // RVALE3OSNXS
when '110 1001 0110 001' return Sys_TLBIP;  // RVAE3NXS
when '110 1001 0110 101' return Sys_TLBIP;  // RVALE3NXS
when '100 1000 0000 010' return Sys_TLBIP;  // RIPAS2E1IS
when '100 1000 0000 110' return Sys_TLBIP;  // RIPAS2LE1IS
when '100 1000 0100 010' return Sys_TLBIP;  // RIPAS2E1
when '100 1000 0100 011' return Sys_TLBIP;  // RIPAS2E1OS
when '100 1000 0100 110' return Sys_TLBIP;  // RIPAS2LE1
when '100 1000 0100 111' return Sys_TLBIP;  // RIPAS2LE1OS
when '100 1001 0000 010' return Sys_TLBIP;  // RIPAS2E1ISNXS
when '100 1001 0000 110' return Sys_TLBIP;  // RIPAS2LE1ISNXS
when '100 1001 0100 010' return Sys_TLBIP;  // RIPAS2E1NXS
when '100 1001 0100 011' return Sys_TLBIP;  // RIPAS2E1OSNXS
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when '100 1001 0100 110' return Sys_TLBIP;  // RIPAS2LE1NXS
        when '100 1001 0100 111' return Sys_TLBIP;  // RIPAS2LE1OSNXS
        otherwise                return Sys_SYSP;
```

G1.1.3.124 SystemOp128

```
// SystemOp128()
// =============
// System instruction types.

enumeration SystemOp128 {Sys_TLBIP, Sys_SYSP};
```

G1.1.3.125 ELR_EL

```
// ELR_EL - getter
// ===============

bits(64) ELR_EL[bits(2) el]
    bits(64) r;
    case el of
        when EL1  r = ELR_EL1;
        when EL2  r = ELR_EL2;
        otherwise Unreachable();
    return r;

// ELR_EL - setter
// ===============

ELR_EL[bits(2) el] = bits(64) value
    constant bits(64) r = value;
    case el of
        when EL1  ELR_EL1 = r;
        when EL2  ELR_EL2 = r;
        otherwise Unreachable();
    return;
```

G1.1.3.126 ELR_ELx

```
// ELR_ELx - getter
// ================

bits(64) ELR_ELx[]
    assert PSTATE.EL != EL0;
    return ELR_EL[PSTATE.EL];

// ELR_ELx - setter
// ================

ELR_ELx[] = bits(64) value
    assert PSTATE.EL != EL0;
    ELR_EL[PSTATE.EL] = value;
    return;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.1.3.127 ESR_EL

```
// ESR_EL - getter
// ===============

ESRType ESR_EL[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1  r = ESR_EL1;
        when EL2  r = ESR_EL2;
        otherwise Unreachable();
    return r;

// ESR_EL - setter
// ===============

ESR_EL[bits(2) regime] = ESRType value
    constant bits(64) r = value;
    case regime of
        when EL1  ESR_EL1 = r;
        when EL2  ESR_EL2 = r;
        otherwise Unreachable();
    return;
```

G1.1.3.128 ESR_ELx

```
// ESR_ELx - getter
// ================

ESRType ESR_ELx[]
    return ESR_EL[S1TranslationRegime()];

// ESR_ELx - setter
// ================

ESR_ELx[] = ESRType value
    ESR_EL[S1TranslationRegime()] = value;
```

G1.1.3.129 FAR_EL

```
// FAR_EL - getter
// ===============

bits(64) FAR_EL[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1  r = FAR_EL1;
        when EL2  r = FAR_EL2;
        otherwise Unreachable();
    return r;

// FAR_EL - setter
// ===============

FAR_EL[bits(2) regime] = bits(64) value
    constant bits(64) r = value;
    case regime of
        when EL1  FAR_EL1 = r;
        when EL2  FAR_EL2 = r;
        otherwise Unreachable();
    return;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.1.3.130 FAR_ELx

```
// FAR_ELx - getter
// ================

bits(64) FAR_ELx[]
    return FAR_EL[S1TranslationRegime()];

// FAR_ELx - setter
// ================

FAR_ELx[] = bits(64) value
    FAR_EL[S1TranslationRegime()] = value;
    return;
```

G1.1.3.131 PRBAR_EL

```
// PRBAR_EL - getter
// =================

PRBARnType PRBAR_EL[bits(2) EL, integer index]
    bits(64) r;
    case EL of
        when EL1  r = PRBARn_EL1[index];
        when EL2  r = PRBARn_EL2[index];
        otherwise Unreachable();
    return r;
```

G1.1.3.132 PRLAR_EL

```
// PRLAR_EL - getter
// =================

PRLARnType PRLAR_EL[bits(2) EL, integer index]
    bits(64) r;
    case EL of
        when EL1  r = PRLARn_EL1[index];
        when EL2  r = PRLARn_EL2[index];
        otherwise Unreachable();
    return r;
```

G1.1.3.133 SCTLR_EL

```
// SCTLR_EL - getter
// =================

SCTLRType SCTLR_EL[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1  r = SCTLR_EL1;
        when EL2  r = SCTLR_EL2;
        otherwise Unreachable();
    return r;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.1.3.134 SCTLR_ELx

```
// SCTLR_ELx - getter
// ==================

SCTLRType SCTLR_ELx[]
    return SCTLR_EL[S1TranslationRegime()];
```

G1.1.3.135 VBAR_EL

```
// VBAR_EL - getter
// ================

bits(64) VBAR_EL[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1  r = VBAR_EL1;
        when EL2  r = VBAR_EL2;
        otherwise Unreachable();
    return r;
```

G1.1.3.136 VBAR_ELx

```
// VBAR_ELx - getter
// =================

bits(64) VBAR_ELx[]
    return VBAR_EL[S1TranslationRegime()];
```

G1.1.3.137 AArch64.CheckDAIFAccess

```
// AArch64.CheckDAIFAccess()
// =========================
// Check that an AArch64 MSR/MRS access to the DAIF flags is permitted.

AArch64.CheckDAIFAccess(PSTATEField field)
    if PSTATE.EL == EL0 && field IN {PSTATEField_DAIFSet, PSTATEField_DAIFClr} then
        if SCTLR_EL1.UMA == '0' then
            if EL2Enabled() && HCR_EL2.TGE == '1' then
                AArch64.SystemAccessTrap(EL2, 0x18);
            else
                AArch64.SystemAccessTrap(EL1, 0x18);
```

G1.1.3.138 AArch64.ImpDefSysInstr

```
// AArch64.ImpDefSysInstr()
// ========================
// Execute an implementation-defined system instruction with write (source operand).

AArch64.ImpDefSysInstr(bits(2) op0, bits(3) op1, bits(4) crn, bits(4) crm, bits(3) op2, integer t);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.1.3.139 AArch64.ImpDefSysInstrWithResult

```
// AArch64.ImpDefSysInstrWithResult()
// ==================================
// Execute an implementation-defined system instruction with read (result operand).

AArch64.ImpDefSysInstrWithResult(bits(2) op0, bits(3) op1, bits(4) crn, bits(4) crm, bits(3) op2,
                                 integer t);
```

G1.1.3.140 AArch64.ImpDefSysRegRead

```
// AArch64.ImpDefSysRegRead()
// ==========================
// Read from an implementation-defined System register and write the contents of the register
// to X[t].

AArch64.ImpDefSysRegRead(bits(2) op0, bits(3) op1, bits(4) crn, bits(4) crm, bits(3) op2,
                         integer t);
```

G1.1.3.141 AArch64.ImpDefSysRegWrite

```
// AArch64.ImpDefSysRegWrite()
// ===========================
// Write to an implementation-defined System register.

AArch64.ImpDefSysRegWrite(bits(2) op0, bits(3) op1, bits(4) crn, bits(4) crm, bits(3) op2,
                          integer t);
```

G1.1.3.142 AArch64.InterruptPending

```
// AArch64.InterruptPending()
// ==========================
// Returns TRUE if there are any pending physical or virtual
// interrupts, and FALSE otherwise.

boolean AArch64.InterruptPending()
    constant boolean pending_physical_interrupt = (IRQPending() || FIQPending() ||
                                                   IsPhysicalSErrorPending());

    boolean pending_virtual_interrupt = FALSE;
    if EL2Enabled() && PSTATE.EL IN {EL0, EL1} && HCR_EL2.TGE == '0' then
        constant boolean virq_pending = (HCR_EL2.IMO == '1' && (VirtualIRQPending() ||
                                         HCR_EL2.VI == '1'));
        constant boolean vfiq_pending = (HCR_EL2.FMO == '1' && (VirtualFIQPending() ||
                                         HCR_EL2.VF == '1'));
        constant boolean vsei_pending = HCR_EL2.AMO == '1' && (IsVirtualSErrorPending() ||
                                                               HCR_EL2.VSE == '1');

        pending_virtual_interrupt = vsei_pending || virq_pending || vfiq_pending;

    return pending_physical_interrupt || pending_virtual_interrupt;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.1.3.143 AArch64.SysInstr

```
// AArch64.SysInstr()
// ==================
// Execute a system instruction with write (source operand).

AArch64.SysInstr(bits(2) op0, bits(3) op1, bits(4) crn, bits(4) crm, bits(3) op2, integer t);
```

G1.1.3.144 AArch64.SysInstrWithResult

```
// AArch64.SysInstrWithResult()
// ============================
// Execute a system instruction with read (result operand).
// Writes the result of the instruction to X[t].

AArch64.SysInstrWithResult(bits(2) op0, bits(3) op1, bits(4) crn, bits(4) crm, bits(3) op2,
                           integer t);
```

G1.1.3.145 AArch64.SysRegRead

```
// AArch64.SysRegRead()
// ====================
// Read from a System register and write the contents of the register to X[t].

AArch64.SysRegRead(bits(2) op0, bits(3) op1, bits(4) crn, bits(4) crm, bits(3) op2, integer t);
```

G1.1.3.146 AArch64.SysRegWrite

```
// AArch64.SysRegWrite()
// =====================
// Write to a System register.

AArch64.SysRegWrite(bits(2) op0, bits(3) op1, bits(4) crn, bits(4) crm, bits(3) op2, integer t);
```

G1.1.3.147 AArch64.TLBI_ALL

```
// AArch64.TLBI_ALL()
// ==================
// Invalidate all entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated broadcast domain.
// Invalidation applies to all applicable stage 1 and stage 2 entries.

AArch64.TLBI_ALL(SecurityState security, Regime regime, Broadcast broadcast_in,
                 TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2};
    Broadcast broadcast = broadcast_in;

    TLBIRecord r;
    r.op          = TLBIOp_ALL;
    r.from_aarch64 = TRUE;
    r.security    = security;
    r.regime      = regime;
    r.level       = TLBILevel_Any;
    r.attr        = attr;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r);
    return;
```

G1.1.3.148 AArch64.TLBI_ASID

```
// AArch64.TLBI_ASID()
// ===================
// Invalidate all stage 1 entries matching the indicated VMID (where regime supports)
// and ASID in the parameter Xt in the indicated translation regime with the
// indicated security state for all TLBs within the indicated broadcast domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch64.TLBI_ASID(SecurityState security, Regime regime, bits(16) vmid,
                  Broadcast broadcast_in, TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};
    Broadcast broadcast = broadcast_in;

    TLBIRecord r;
    r.op           = TLBIOp_ASID;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = UseVMID(regime);
    r.level        = TLBILevel_Any;
    r.attr         = attr;
    r.asid         = Xt<63:48>;

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r);

    return;
```

G1.1.3.149 AArch64.TLBI_IPAS2

```
// AArch64.TLBI_IPAS2()
// ====================
// Invalidate by IPA all stage 2 only TLB entries in the indicated broadcast
// domain matching the indicated VMID in the indicated regime with the indicated security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of this operation.
// IPA and related parameters of the are derived from Xt.

AArch64.TLBI_IPAS2(SecurityState security, Regime regime, bits(16) vmid,
                   Broadcast broadcast_in, TLBILevel level, TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2};
    Broadcast broadcast = broadcast_in;

    TLBIRecord r;
    r.op           = TLBIOp_IPAS2;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = TRUE;
    r.level        = level;
    r.attr         = attr;
    r.ttl          = Xt<47:44>;
    r.address      = ZeroExtend(Xt<39:0> : Zeros(12), 64);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    case security of
        when SS_NonSecure
            r.ipaspace = PAS_NonSecure;
        when SS_Secure
            r.ipaspace = if Xt<63> == '1' then PAS_NonSecure else PAS_Secure;

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r);

    return;
```

G1.1.3.150 AArch64.TLBI_RIPAS2

```
// AArch64.TLBI_RIPAS2()
// =====================
// Range invalidate by IPA all stage 2 only TLB entries in the indicated
// broadcast domain matching the indicated VMID in the indicated regime with the indicated
// security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of this operation.
// The range of IPA and related parameters of the are derived from Xt.

AArch64.TLBI_RIPAS2(SecurityState security, Regime regime, bits(16) vmid,
                    Broadcast broadcast_in, TLBILevel level, TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};
    Broadcast broadcast = broadcast_in;

    TLBIRecord r;
    r.op           = TLBIOp_RIPAS2;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = TRUE;
    r.level        = level;
    r.attr         = attr;
    r.ttl<1:0>     = Xt<38:37>;

    boolean valid;
    (valid, r.tg, r.address, r.end_address) = TLBIRange(regime, Xt);

    if !valid then return;

    case security of
        when SS_NonSecure
            r.ipaspace = PAS_NonSecure;
        when SS_Secure
            r.ipaspace = if Xt<63> == '1' then PAS_NonSecure else PAS_Secure;

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r);
    return;
```

G1.1.3.151 AArch64.TLBI_RVA

```
// AArch64.TLBI_RVA()
// ==================
// Range invalidate by VA range all stage 1 TLB entries in the indicated
// broadcast domain matching the indicated VMID and ASID (where regime
// supports VMID, ASID) in the indicated regime with the indicated security state.
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// ASID, and range related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch64.TLBI_RVA(SecurityState security, Regime regime, bits(16) vmid,
                 Broadcast broadcast_in, TLBILevel level, TLBIMemAttr attr, bits(64)  Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};
    Broadcast broadcast = broadcast_in;

    TLBIRecord r;
    r.op           = TLBIOp_RVA;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = UseVMID(regime);
    r.level        = level;
    r.attr         = attr;
    r.asid         = Xt<63:48>;
    r.ttl<1:0>     = Xt<38:37>;

    boolean valid;
    (valid, r.tg, r.address, r.end_address) = TLBIRange(regime, Xt);

    if !valid then return;

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r);
    return;
```

G1.1.3.152 AArch64.TLBI_RVAA

```
// AArch64.TLBI_RVAA()
// ===================
// Range invalidate by VA range all stage 1 TLB entries in the indicated
// broadcast domain matching the indicated VMID (where regimesupports VMID)
// and all ASID in the indicated regime with the indicated security state.
// VA range related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch64.TLBI_RVAA(SecurityState security, Regime  regime, bits(16) vmid,
                  Broadcast broadcast, TLBILevel level, TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_RVAA;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = UseVMID(regime);
    r.level        = level;
    r.attr         = attr;
    r.ttl<1:0>     = Xt<38:37>;

    boolean valid;
    (valid, r.tg, r.address, r.end_address) = TLBIRange(regime, Xt);

    if !valid then return;

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r);
    return;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.1.3.153 AArch64.TLBI_VA

```
// AArch64.TLBI_VA()
// =================
// Invalidate by VA all stage 1 TLB entries in the indicated broadcast domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// ASID, VA and related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch64.TLBI_VA(SecurityState security, Regime regime, bits(16) vmid,
                Broadcast broadcast_in, TLBILevel level,  TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};
    Broadcast broadcast = broadcast_in;

    TLBIRecord r;
    r.op           = TLBIOp_VA;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = UseVMID(regime);
    r.level        = level;
    r.attr         = attr;
    r.asid         = Xt<63:48>;
    r.ttl          = Xt<47:44>;
    r.address      = ZeroExtend(Xt<43:0> : Zeros(12), 64);

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r);

    return;
```

G1.1.3.154 AArch64.TLBI_VAA

```
// AArch64.TLBI_VAA()
// ==================
// Invalidate by VA all stage 1 TLB entries in the indicated broadcast domain
// matching the indicated VMID (where regime supports VMID) and all ASID in the indicated regime
// with the indicated security state.
// VA and related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch64.TLBI_VAA(SecurityState security, Regime regime, bits(16) vmid,
                 Broadcast broadcast_in, TLBILevel level, TLBIMemAttr attr,  bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};
    Broadcast broadcast = broadcast_in;

    TLBIRecord r;
    r.op           = TLBIOp_VAA;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = UseVMID(regime);
    r.level        = level;
    r.attr         = attr;
    r.ttl          = Xt<47:44>;
    r.address      = ZeroExtend(Xt<43:0> : Zeros(12), 64);

    TLBI(r);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r);

    return;
```

G1.1.3.155 AArch64.TLBI_VMALL

```
// AArch64.TLBI_VMALL()
// ====================
// Invalidate all stage 1 entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated broadcast
// domain that match the indicated VMID (where applicable).
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// Note: stage 2 only entries are not in the scope of this operation.

AArch64.TLBI_VMALL(SecurityState security, Regime regime, bits(16) vmid,
                   Broadcast broadcast_in, TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};
    Broadcast broadcast = broadcast_in;

    TLBIRecord r;
    r.op           = TLBIOp_VMALL;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.level        = TLBILevel_Any;
    r.vmid         = vmid;
    r.use_vmid     = UseVMID(regime);
    r.attr         = attr;

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r);

    return;
```

G1.1.3.156 AArch64.TLBI_VMALLS12

```
// AArch64.TLBI_VMALLS12()
// =======================
// Invalidate all stage 1 and stage 2 entries for the indicated translation
// regime with the indicated security state for all TLBs within the indicated
// broadcast domain that match the indicated VMID.

AArch64.TLBI_VMALLS12(SecurityState security, Regime regime, bits(16) vmid,
                      Broadcast broadcast_in, TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2};
    Broadcast broadcast = broadcast_in;

    TLBIRecord r;
    r.op           = TLBIOp_VMALLS12;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.level        = TLBILevel_Any;
    r.vmid         = vmid;
    r.use_vmid     = TRUE;
    r.attr         = attr;

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r);

    return;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.1.3.157 DecodeTLBITG

```
// DecodeTLBITG()
// ==============
// Decode translation granule size in TLBI range instructions

TGx DecodeTLBITG(bits(2) tg)
    case tg of
        when '01'   return TGx_4KB;
        when '10'   return TGx_16KB;
        when '11'   return TGx_64KB;
```

G1.1.3.158 HasLargeAddress

```
// HasLargeAddress()
// =================
// Returns TRUE if the regime is configured for 52 bit addresses, FALSE otherwise.

boolean HasLargeAddress(Regime regime)
    return FALSE;
```

G1.1.3.159 ResTLBIRTTL

```
// ResTLBIRTTL()
// =============
// Determine whether the TTL field in TLBI instructions that do apply
// to a range of addresses contains a reserved value

boolean ResTLBIRTTL(bits(2) tg, bits(2) ttl)
    case ttl of
        when '00' return TRUE;
        when '01' return DecodeTLBITG(tg) == TGx_16KB;
        otherwise return FALSE;
```

G1.1.3.160 ResTLBITTL

```
// ResTLBITTL()
// ============
// Determine whether the TTL field in TLBI instructions that do not apply
// to a range of addresses contains a reserved value

boolean ResTLBITTL(bits(4) ttl)
    case ttl of
        when '00xx' return TRUE;
        when '0100' return TRUE;
        when '1000' return TRUE;
        when '1001' return TRUE;
        when '1100' return TRUE;
        otherwise   return FALSE;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.1.3.161 TLBIMatch

```
// TLBIMatch()
// ===========
// Determine whether the TLB entry lies within the scope of invalidation

boolean TLBIMatch(TLBIRecord tlbi, TLBRecord tlb_entry)
    boolean match;
    constant bits(64) entry_block_mask    = ZeroExtend(Ones(tlb_entry.blocksize), 64);
    bits(64) entry_end_address   = tlb_entry.context.ia OR entry_block_mask;
    bits(64) entry_start_address = tlb_entry.context.ia AND NOT entry_block_mask;
    case tlbi.op of
        when TLBIOp_ALL
            match = (tlbi.security == tlb_entry.context.ss &&
                     tlbi.regime  == tlb_entry.context.regime);
        when TLBIOp_ASID
            match = (tlb_entry.context.includes_s1 &&
                     tlbi.security == tlb_entry.context.ss &&
                     tlbi.regime   == tlb_entry.context.regime &&
                     tlbi.use_vmid == tlb_entry.context.use_vmid &&
                     (!tlb_entry.context.use_vmid || tlbi.vmid == tlb_entry.context.vmid) &&
                     (UseASID(tlb_entry.context) && tlb_entry.context.nG == '1' &&
                        tlbi.asid  == tlb_entry.context.asid));
        when TLBIOp_IPAS2
            constant integer addr_lsb = tlb_entry.blocksize;
            match = (!tlb_entry.context.includes_s1 && tlb_entry.context.includes_s2 &&
                     tlbi.security == tlb_entry.context.ss &&
                     tlbi.regime   == tlb_entry.context.regime &&
                     (!tlb_entry.context.use_vmid || tlbi.vmid == tlb_entry.context.vmid) &&
                     tlbi.ipaspace == tlb_entry.context.ipaspace &&
                     tlbi.address<51:addr_lsb> == tlb_entry.context.ia<51:addr_lsb> &&
                     (!tlbi.from_aarch64 || ResTLBITTL(tlbi.ttl) || (
                         DecodeTLBITG(tlbi.ttl<3:2>) == tlb_entry.context.tg &&
                         UInt(tlbi.ttl<1:0>) == tlb_entry.walkstate.level)
                     ) &&
                     (tlbi.level == TLBILevel_Any || !tlb_entry.walkstate.istable));
        when TLBIOp_VAA
            constant integer addr_lsb = tlb_entry.blocksize;
            match = (tlb_entry.context.includes_s1 &&
                     tlbi.security == tlb_entry.context.ss &&
                     tlbi.regime   == tlb_entry.context.regime &&
                     tlbi.use_vmid == tlb_entry.context.use_vmid &&
                     (!tlb_entry.context.use_vmid || tlbi.vmid == tlb_entry.context.vmid) &&
                     tlbi.address<55:addr_lsb> == tlb_entry.context.ia<55:addr_lsb> &&
                     (!tlbi.from_aarch64 || ResTLBITTL(tlbi.ttl) || (
                         DecodeTLBITG(tlbi.ttl<3:2>) == tlb_entry.context.tg &&
                         UInt(tlbi.ttl<1:0>) == tlb_entry.walkstate.level)
                     ) &&
                     (tlbi.level == TLBILevel_Any || !tlb_entry.walkstate.istable));
        when TLBIOp_VA
            constant integer addr_lsb = tlb_entry.blocksize;
            match = (tlb_entry.context.includes_s1 &&
                     tlbi.security == tlb_entry.context.ss &&
                     tlbi.regime   == tlb_entry.context.regime &&
                     tlbi.use_vmid == tlb_entry.context.use_vmid &&
                     (!tlb_entry.context.use_vmid || tlbi.vmid == tlb_entry.context.vmid) &&
                     (!UseASID(tlb_entry.context) || tlbi.asid == tlb_entry.context.asid ||
                        tlb_entry.context.nG == '0') &&
                     tlbi.address<55:addr_lsb> == tlb_entry.context.ia<55:addr_lsb> &&
                     (!tlbi.from_aarch64 || ResTLBITTL(tlbi.ttl) || (
                         DecodeTLBITG(tlbi.ttl<3:2>) == tlb_entry.context.tg &&
                         UInt(tlbi.ttl<1:0>) == tlb_entry.walkstate.level)
                     ) &&
                     (tlbi.level == TLBILevel_Any || !tlb_entry.walkstate.istable));
        when TLBIOp_VMALL
            match = (tlb_entry.context.includes_s1 &&
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                    tlbi.security == tlb_entry.context.ss &&
                    tlbi.regime   == tlb_entry.context.regime &&
                    tlbi.use_vmid == tlb_entry.context.use_vmid &&
                    (!tlb_entry.context.use_vmid || tlbi.vmid == tlb_entry.context.vmid));
        when TLBIOp_VMALLS12
            match = (tlbi.security == tlb_entry.context.ss &&
                    tlbi.regime   == tlb_entry.context.regime &&
                    (!tlb_entry.context.use_vmid || tlbi.vmid == tlb_entry.context.vmid));
        when TLBIOp_RIPAS2
            match = (!tlb_entry.context.includes_s1 && tlb_entry.context.includes_s2 &&
                    tlbi.security == tlb_entry.context.ss &&
                    tlbi.regime   == tlb_entry.context.regime &&
                    (!tlb_entry.context.use_vmid || tlbi.vmid == tlb_entry.context.vmid) &&
                    tlbi.ipaspace == tlb_entry.context.ipaspace &&
                    (tlbi.tg != '00' && DecodeTLBITG(tlbi.tg) == tlb_entry.context.tg) &&
                    (!tlbi.from_aarch64 || ResTLBIRTTL(tlbi.tg, tlbi.ttl<1:0>) ||
                       UInt(tlbi.ttl<1:0>) == tlb_entry.walkstate.level) &&
                    UInt(tlbi.address<51:0>) <= UInt(entry_end_address<51:0>) &&
                    UInt(tlbi.end_address<51:0>) > UInt(entry_start_address<51:0>));
        when TLBIOp_RVAA
            match = (tlb_entry.context.includes_s1 &&
                    tlbi.security == tlb_entry.context.ss &&
                    tlbi.regime   == tlb_entry.context.regime &&
                    tlbi.use_vmid == tlb_entry.context.use_vmid &&
                    (!tlb_entry.context.use_vmid || tlbi.vmid == tlb_entry.context.vmid) &&
                    (tlbi.tg != '00' && DecodeTLBITG(tlbi.tg) == tlb_entry.context.tg) &&
                    (!tlbi.from_aarch64 || ResTLBIRTTL(tlbi.tg, tlbi.ttl<1:0>) ||
                       UInt(tlbi.ttl<1:0>) == tlb_entry.walkstate.level) &&
                    UInt(tlbi.address<55:0>) <= UInt(entry_end_address<55:0>) &&
                    UInt(tlbi.end_address<55:0>) > UInt(entry_start_address<55:0>));
        when TLBIOp_RVA
            match = (tlb_entry.context.includes_s1 &&
                    tlbi.security == tlb_entry.context.ss &&
                    tlbi.regime   == tlb_entry.context.regime &&
                    tlbi.use_vmid == tlb_entry.context.use_vmid &&
                    (!tlb_entry.context.use_vmid || tlbi.vmid == tlb_entry.context.vmid) &&
                    (!UseASID(tlb_entry.context) || tlbi.asid == tlb_entry.context.asid ||
                       tlb_entry.context.nG == '0') &&
                    (tlbi.tg != '00' && DecodeTLBITG(tlbi.tg) == tlb_entry.context.tg) &&
                    (!tlbi.from_aarch64 || ResTLBIRTTL(tlbi.tg, tlbi.ttl<1:0>) ||
                       UInt(tlbi.ttl<1:0>) == tlb_entry.walkstate.level) &&
                    UInt(tlbi.address<55:0>) <= UInt(entry_end_address<55:0>) &&
                    UInt(tlbi.end_address<55:0>) > UInt(entry_start_address<55:0>));

    return match;
```

G1.1.3.162 TLBIRange

```
// TLBIRange()
// ===========
// Extract the input address range information from encoded Xt.

(boolean, bits(2), bits(64), bits(64)) TLBIRange(Regime regime, bits(64) Xt)
    constant boolean  valid = TRUE;
    bits(64) start_address = Zeros(64);
    bits(64) end_address   = Zeros(64);

    constant bits(2) tg    = Xt<47:46>;
    constant integer scale = UInt(Xt<45:44>);
    constant integer num   = UInt(Xt<43:39>);
    integer tg_bits;

    if tg == '00' then
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        return (FALSE, tg, start_address, end_address);

    case tg of
        when '01' // 4KB
            tg_bits = 12;
            if HasLargeAddress(regime) then
                start_address<52:16> = Xt<36:0>;
                start_address<63:53> = Replicate(Xt<36>, 11);
            else
                start_address<48:12> = Xt<36:0>;
                start_address<63:49> = Replicate(Xt<36>, 15);
        when '10' // 16KB
            tg_bits = 14;
            if HasLargeAddress(regime) then
                start_address<52:16> = Xt<36:0>;
                start_address<63:53> = Replicate(Xt<36>, 11);
            else
                start_address<50:14> = Xt<36:0>;
                start_address<63:51> = Replicate(Xt<36>, 13);
        when '11' // 64KB
            tg_bits = 16;
            start_address<52:16> = Xt<36:0>;
            start_address<63:53> = Replicate(Xt<36>, 11);
        otherwise
            Unreachable();

    constant integer range = (num+1) << (5*scale + 1 + tg_bits);
    end_address   = start_address + range<63:0>;

    if end_address<52> != start_address<52> then
        // overflow, saturate it
        end_address = Replicate(start_address<52>, 12) : Ones(52);

    return (valid, tg, start_address, end_address);
```

G1.1.3.163 VBitOp

```
// VBitOp
// ======
// Vector bit select instruction types.

enumeration VBitOp      {VBitOp_VBIF, VBitOp_VBIT, VBitOp_VBSL, VBitOp_VEOR};
```

G1.1.4 aarch64/translation

This section includes the following pseudocode functions:

G1.1.4.1 AArch64.MAIRAttr

```
// AArch64.MAIRAttr()
// ==================
// Retrieve the memory attribute encoding indexed in the given MAIR

bits(8) AArch64.MAIRAttr(integer index, MAIRType mair)
    assert (index < 8);
    return Elem[mair, index, 8];
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.1.4.2 AArch64.CheckBreakpoint

```
// AArch64.CheckBreakpoint()
// =========================
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch64
// translation regime, when either debug exceptions are enabled, or halting debug is enabled
// and halting is allowed.

FaultRecord AArch64.CheckBreakpoint(FaultRecord fault_in, bits(64) vaddress,
                                    AccessDescriptor accdesc, integer size)
    assert !ELUsingAArch32(S1TranslationRegime());
    assert (UsingAArch32() && size IN {2,4}) || size == 4;

    FaultRecord fault = fault_in;
    boolean match = FALSE;

    for i = 0 to NumBreakpointsImplemented() - 1
        constant BreakpointInfo brkptinfo = AArch64.BreakpointMatch(i, vaddress, accdesc, size);
        match = match || brkptinfo.match;

    if match then
        fault.statuscode = Fault_Debug;
        fault.vaddress   = vaddress;
        if HaltOnBreakpointOrWatchpoint() then
            reason = DebugHalt_Breakpoint;
            Halt(reason);

    return fault;
```

G1.1.4.3 AArch64.CheckDebug

```
// AArch64.CheckDebug()
// ====================
// Called on each access to check for a debug exception or entry to Debug state.

FaultRecord AArch64.CheckDebug(bits(64) vaddress, AccessDescriptor accdesc, integer size)

    FaultRecord fault = NoFault(accdesc, vaddress);
    boolean generate_exception;

    constant boolean d_side = IsWatchpointableAccess(accdesc);
    constant boolean i_side = (accdesc.acctype == AccessType_IFETCH);
    generate_exception = AArch64.GenerateDebugExceptions() && MDSCR_EL1.MDE == '1';
    halt = HaltOnBreakpointOrWatchpoint();

    if generate_exception || halt then
        if d_side then
            fault = AArch64.CheckWatchpoint(fault, vaddress, accdesc, size);
        elsif i_side then
            fault = AArch64.CheckBreakpoint(fault, vaddress, accdesc, size);

    return fault;
```

G1.1.4.4 AArch64.CheckWatchpoint

```
// AArch64.CheckWatchpoint()
// =========================
// Called before accessing the memory location of "size" bytes at "address",
// when either debug exceptions are enabled for the access, or halting debug
// is enabled and halting is allowed.
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
FaultRecord AArch64.CheckWatchpoint(FaultRecord fault_in, bits(64) vaddress_in,
                                    AccessDescriptor accdesc, integer size_in)
    assert !ELUsingAArch32(S1TranslationRegime());
    FaultRecord fault          = fault_in;
    bits(64) vaddress          = vaddress_in;
    integer size               = size_in;

    for i = 0 to NumWatchpointsImplemented() - 1
        constant WatchpointInfo watchptinfo = AArch64.WatchpointMatch(i, vaddress, size, accdesc);
        if watchptinfo.value_match then
            fault.statuscode = Fault_Debug;
            if DBGWCR_EL1[i].LSC<0> == '1' && accdesc.read then
                fault.write = FALSE;
            elsif DBGWCR_EL1[i].LSC<1> == '1' && accdesc.write then
                fault.write = TRUE;
            fault.watchptinfo = watchptinfo;
    fault.vaddress = vaddress;
    if fault.statuscode == Fault_Debug && HaltOnBreakpointOrWatchpoint() then
        reason = DebugHalt_Watchpoint;
        EDWAR = fault.vaddress;
        is_async = FALSE;
        Halt(reason, is_async);
    return fault;
```

G1.1.4.5 AArch64.DetermineS2PASpace

```
// AArch64.DetermineS2PASpace()
// ============================
// Determine stage 2 Physical Address Space for EL1&0 translation regime.

PASpace AArch64.DetermineS2PASpace(PASpace s1paspace, PASpace s2paspace)
    if s1paspace == PAS_Secure && s2paspace == PAS_Secure then
        if VSTCR_EL2.SA == '1' then
            return PAS_NonSecure;
        else
            return PAS_Secure;
    else
        if VSTCR_EL2.SA == '1' || VTCR_EL2.NSA == '1' then
            return PAS_NonSecure;
        else
            return PAS_Secure;
```

G1.1.4.6 AArch64.FullValidate

```
// AArch64.FullValidate()
// ======================
// Apply VMSA translation / PMSA validation on memory access subject to
// configuration and translation regime

AddressDescriptor AArch64.FullValidate(bits(64) va, integer size, AccessDescriptor accdesc,
                                       boolean aligned)
    AddressDescriptor ipa;
    AddressDescriptor pa;
    // Initialise fault record in case a fault is detected down the line
    fault = NoFault(accdesc, va);
    regime = TranslationRegime(PSTATE.EL);

    if AArch64.IsStage1VMSA(regime) then
        // Stage 1 translation follows VMSAv8-64
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        (fault, ipa) = AArch64.S1Translate(fault, regime, va, size, aligned, accdesc);
    else
        // Stage 1 is validation follows PMSAv8-64
        (fault, ipa) = AArch64.S1Validate(fault, regime, va, aligned, accdesc);

    if fault.statuscode != Fault_None then
        return CreateFaultyAddressDescriptor(fault);

    // Second Stage Validation
    if regime == Regime_EL10 then
        (fault, pa) = AArch64.S2Validate(fault, ipa, aligned, accdesc);

        if fault.statuscode != Fault_None then
            return CreateFaultyAddressDescriptor(fault);
        return pa;
    else
        return ipa;
```

G1.1.4.7 AArch64.IsStage1VMSA

```
// AArch64.IsStage1VMSA()
// ======================
// Determine whether V8A VMSA is applied to stage 1 translation

boolean AArch64.IsStage1VMSA(Regime regime)
    return regime == Regime_EL10 && HaveEL1VMSAExt() && VTCR_EL2.MSA == '1';
```

G1.1.4.8 AArch64.MPUValidate

```
// AArch64.MPUValidate()
// =====================
// Attempt to match the input address with an active Memory Protection Unit (MPU) and
// retrieve assigned permissions and memory attributes

(FaultRecord, boolean, PRBARnType, PRLARnType) AArch64.MPUValidate(FaultRecord fault_in,
                                                                   Regime regime, bits(64) ia)
    FaultRecord fault = fault_in;
    integer num_regions;
    PRBARnType matched_prbar;
    PRLARnType matched_prlar;
    matched = FALSE;
    case regime of
        when Regime_EL2  num_regions = SInt(MPUIR_EL2.REGION);
        when Regime_EL10 num_regions = SInt(MPUIR_EL1.REGION);

    for index = 0 to num_regions-1
        PRBARnType prbar;
        PRLARnType prlar;
        case regime of
            when Regime_EL2
                prbar = PRBAR_EL[EL2, index];
                prlar = PRLAR_EL[EL2, index];
            when Regime_EL10
                prbar = PRBAR_EL[EL1, index];
                prlar = PRLAR_EL[EL1, index];

        baseaddr  = ZeroExtend(prbar.BASE  : Zeros(6), 64);
        limitaddr = ZeroExtend(prlar.LIMIT :  Ones(6), 64);

        // Check for a matching MPU region
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if (prlar.EN == '1' &&
                UInt(ia) >= UInt(baseaddr) &&
                UInt(ia) <= UInt(limitaddr)) then

            // Check for multiple region match
            if matched then
                fault.statuscode = Fault_Translation;
                return (fault, matched, PRBARnType UNKNOWN, PRLARnType UNKNOWN);

            matched = TRUE;
            matched_prbar = prbar;
            matched_prlar = prlar;

    return (fault, matched, matched_prbar, matched_prlar);
```

G1.1.4.9 AArch64.S1Validate

```
// AArch64.S1Validate()
// ====================
// Perform stage 1 PMSA validation using Memory Protection Units (MPUs).

(FaultRecord, AddressDescriptor) AArch64.S1Validate(FaultRecord fault_in, Regime regime,
                                                    bits(64) va_in, boolean aligned,
                                                    AccessDescriptor accdesc)
    FaultRecord fault = fault_in;
    bits(64) va = va_in;
    // Prepare fault fields if one is detected
    fault.secondstage = FALSE;
    fault.s2fs1walk   = FALSE;
    fault.level       = 0;

    if !(AArch64.S1BREnabled(regime) || AArch64.S1Enabled(regime, accdesc.acctype)) then
        return AArch64.S1DisabledOutput(fault, regime, va, accdesc, aligned);

    constant bit tbid = if regime == Regime_EL2 then TCR_EL2.TBID else TCR_EL1.TBID0;
    constant bit tbi  = if regime == Regime_EL2 then TCR_EL2.TBI  else TCR_EL1.TBI0;

    constant AddressSize addrtop = AArch64.AddrTop(tbid, accdesc.acctype, tbi);
    constant AddressSize pamax = AArch64.PAMax();
    if !IsZero(va<addrtop:pamax>) then
        fault.statuscode = Fault_Translation;
        return (fault, AddressDescriptor UNKNOWN);

    // Clear out bits not part of MPU matching
    va = ZeroExtend(va<addrtop:0>, 64);

    Permissions s1_permissions;
    PASpace s1_paspace;
    bits(8) s1_attr;
    bits(2) s1_sh;
    boolean valid;
    if AArch64.S1Enabled(regime, accdesc.acctype) then
        boolean matched;
        PRBARnType prbar;
        PRLARnType prlar;
        (fault, matched, prbar, prlar) = AArch64.MPUValidate(fault, regime, va);

        if fault.statuscode != Fault_None then
            return (fault, AddressDescriptor UNKNOWN);
        elsif matched then
            s1_paspace = if prlar.NS == '0' then PAS_Secure else PAS_NonSecure;

            if HasUnprivileged(regime) then
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            s1_permissions.ap<2:1> = prbar.AP;
            s1_permissions.pxn     = prbar.XN<1>;
            s1_permissions.uxn     = prbar.XN<1>;
        else
            s1_permissions.ap<2:1> = prbar.AP<1>:'1';
            s1_permissions.xn      = prbar.XN<1>;

        MAIRType s1_mair;
        case regime of
            when Regime_EL2  s1_mair = MAIR_EL2;
            when Regime_EL10 s1_mair = MAIR_EL1;

        s1_attr = AArch64.MAIRAttr(UInt(prlar.AttrIndx), s1_mair);
        s1_sh   = prbar.SH;

    elsif AArch64.S1BREnabled(regime) && accdesc.el != EL0 then
        assert (boolean IMPLEMENTATION_DEFINED "Default Memory Map");
        (valid, s1_attr, s1_sh, s1_permissions,
            s1_paspace) = __BackgroundMemoryAttr(va, accdesc);

        if !valid then
            fault.statuscode = Fault_Translation;
            return (fault, AddressDescriptor UNKNOWN);
    else
        // No MPU match nor background region enabled
        fault.statuscode = Fault_Translation;
        return (fault, AddressDescriptor UNKNOWN);

elsif AArch64.S1BREnabled(regime) then
    assert (boolean IMPLEMENTATION_DEFINED "Default Memory Map");
    (valid, s1_attr, s1_sh, s1_permissions,
        s1_paspace) = __BackgroundMemoryAttr(va, accdesc);

    if !valid then
        fault.statuscode = Fault_Translation;
        return (fault, AddressDescriptor UNKNOWN);

MPURecord mpurecord;
mpurecord.paspace     = s1_paspace;
mpurecord.permissions = s1_permissions;
s1aarch64 = TRUE;
mpurecord.memattrs    = S1DecodeMemAttrs(s1_attr, s1_sh, s1aarch64);

if AArch64.S1HasAlignmentFaultDueToMemType(regime, accdesc, aligned, mpurecord.memattrs) then
    fault.statuscode = Fault_Alignment;
    return (fault, AddressDescriptor UNKNOWN);

fault = AArch64.S1CheckPermissions_PMSA(fault, regime, mpurecord.memattrs,
                                        mpurecord.permissions, accdesc);

if fault.statuscode != Fault_None then
    return (fault, AddressDescriptor UNKNOWN);

MemoryAttributes memattrs;
if ((accdesc.acctype == AccessType_IFETCH &&
       (mpurecord.memattrs.memtype == MemType_Device || !AArch64.S1ICacheEnabled(regime))) ||
    (accdesc.acctype != AccessType_IFETCH &&
        mpurecord.memattrs.memtype == MemType_Normal && !AArch64.S1DCacheEnabled(regime))) then
    // Treat memory attributes as Normal Non-Cacheable
    memattrs = NormalNCMemAttr();
else
    memattrs = mpurecord.memattrs;

// Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
// to be either effective value or descriptor value
if (regime == Regime_EL10 && HCR_EL2.VM == '1' &&
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1")) then
        mematttrs.shareability = mpurecord.mematttrs.shareability;
    else
        mematttrs.shareability = EffectiveShareability(mematttrs);

    // Output Address
    FullAddress oa;
    oa.paspace = mpurecord.paspace;
    oa.address = va<51:0>;
    ipa = CreateAddressDescriptor(va, oa, mematttrs, accdesc);
    return (fault, ipa);
```

G1.1.4.10 AArch64.S2Validate

```
// AArch64.S2Validate()
// ====================
// Perform stage 2 PMSA validation using Memory Protection Units (MPUs).

(FaultRecord, AddressDescriptor) AArch64.S2Validate(FaultRecord fault_in, AddressDescriptor ipa,
                                                    boolean aligned, AccessDescriptor accdesc)

    FaultRecord fault = fault_in;
    // Prepare fault fields in case a fault is detected
    fault.statuscode  = Fault_None; // Ignore any faults from stage 1
    fault.secondstage = TRUE;
    fault.s2fs1walk   = accdesc.acctype == AccessType_TTW;
    fault.level       = 0;
    fault.ipaddress   = ipa.paddress;

    PASpace s2_paspace;
    Permissions s2_permissions;
    bits(8) s2_attr;
    bits(2) s2_sh;
    boolean valid;
    AddressDescriptor pa;
    ipa_64 = ZeroExtend(ipa.paddress.address, 64);

    // Stage 2 is disabled
    if HCR_EL2.DC == '0' && HCR_EL2.VM == '0' then
        pa = ipa;
        return (fault, pa);

    if SCTLR_EL2.M == '1' then
        boolean matched;
        PRBARnType prbar;
        PRLARnType prlar;
        // Stage 2 set of MPUs are the exact same ones as those used for stage 1 EL2 regime
        (fault, matched, prbar, prlar) = AArch64.MPUValidate(fault, Regime_EL2, ipa_64);

        if fault.statuscode != Fault_None then
            return (fault, AddressDescriptor UNKNOWN);
        elsif matched then
            s2_paspace = if prlar.NS == '0' then PAS_Secure else PAS_NonSecure;

            if VSTCR_EL2.SC == '1' && ipa.paddress.paspace != s2_paspace then
                fault.statuscode = Fault_Translation;
                return (fault, AddressDescriptor UNKNOWN);

            s2_permissions.ap<2:1> = prbar.AP;
            s2_permissions.xn      = prbar.XN<1>;
            if IsFeatureImplemented(FEAT_XNX) then
                s2_permissions.s2xnx = prbar.XN<0>;
            else
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            s2_permissions.s2xnx = '0';

        s2_attr = AArch64.MAIRAttr(UInt(prlar.AttrIndx), MAIR_EL2);
        s2_sh   = prbar.SH;
    else
        // No MPU match
        fault.statuscode = Fault_Translation;
        return (fault, AddressDescriptor UNKNOWN);

elsif SCTLR_EL2.BR == '1' then
    assert (boolean IMPLEMENTATION_DEFINED "Default Memory Map");

    (valid, s2_attr, s2_sh, s2_permissions,
        s2_paspace) = __BackgroundMemoryAttr(ipa_64, accdesc);

    if !valid then
        fault.statuscode = Fault_Translation;
        return (fault, AddressDescriptor UNKNOWN);
else
    // When HCR_EL2.VM is effectively '1' and SCTLR_EL2.{M, BR} = {0,0},
    // the behavior is CONSTRAINED UNPREDICTABLE.
    c = ConstrainUnpredictable(Unpredictable_S2EL1MPUCONFIG);
    assert c IN {Constraint_MPU_FAULT, Constraint_MPU_ATTR_UNKNOWN};

    if c == Constraint_MPU_FAULT then
        fault.statuscode = Fault_Translation;
        return (fault, AddressDescriptor UNKNOWN);
    else
        s2_attr        = bits(8) UNKNOWN;
        s2_sh          = bits(2) UNKNOWN;
        s2_permissions = Permissions UNKNOWN;
        s2_paspace     = ipa.paddress.paspace;

MPURecord mpurecord;
// Stage 2 PA space is determined from stage 2 configuration and stage 1 IPA space
mpurecord.paspace     = AArch64.DetermineS2PASpace(ipa.paddress.paspace, s2_paspace);
mpurecord.permissions = s2_permissions;

if IsFeatureImplemented(FEAT_S2FWB) && HCR_EL2.FWB == '1' then
    mpurecord.memattrs = AArch64.S2ApplyFWBMemAttrs(ipa.memattrs, s2_attr, s2_sh);
else
    // V8R stage 2 memory attributes are decoded in the same fashion as stage 1
    // for the EL2 regime. The only difference is allocation hints are ignored
    s1aarch64 = TRUE;
    mpurecord.memattrs = S1DecodeMemAttrs(s2_attr, s2_sh, s1aarch64);

if IsFeatureImplemented(FEAT_TTCNP) && AArch64.IsStage1VMSA(Regime_EL10) then
    mpurecord.CnP = VSCTLR_EL2.CnP;

if AArch64.S2HasAlignmentFaultDueToMemType(accdesc, aligned, mpurecord.memattrs) then
    fault.statuscode = Fault_Alignment;
    return (fault, AddressDescriptor UNKNOWN);

fault = AArch64.S2CheckPermissions(fault, mpurecord.memattrs, mpurecord.permissions, accdesc);

if fault.statuscode != Fault_None then
    return (fault, AddressDescriptor UNKNOWN);

MemoryAttributes s2_memattrs;
if ((accdesc.acctype == AccessType_TTW &&
        mpurecord.memattrs.memtype == MemType_Device &&  HCR_EL2.PTW == '0') ||
    (accdesc.acctype == AccessType_IFETCH &&
        (mpurecord.memattrs.memtype == MemType_Device || HCR_EL2.ID == '1')) ||
    (accdesc.acctype != AccessType_IFETCH &&
         mpurecord.memattrs.memtype == MemType_Normal && !S2DCacheEnabled())) then
    // Treat memory attributes as Normal Non-Cacheable
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        s2_mematttrs = NormalNCMemAttr();
    else
        s2_mematttrs = mpurecord.mematttrs;

    s2aarch64 = TRUE;
    MemoryAttributes mematttrs;
    if !IsFeatureImplemented(FEAT_S2FWB) || HCR_EL2.FWB == '0' then
        mematttrs = S2CombineS1MemAttrs(ipa.mematttrs, s2_mematttrs, s2aarch64);
    else
        mematttrs = s2_mematttrs;

    // Output Address
    FullAddress oa;
    oa.paspace = mpurecord.paspace;
    oa.address = ipa.paddress.address;
    pa = CreateAddressDescriptor(ipa.vaddress, oa, mematttrs, accdesc);
    return (fault, pa);
```

G1.1.4.11 AArch64.IASize

```
// AArch64.IASize()
// ================
// Retrieve the number of bits containing the input address

AddressSize AArch64.IASize(bits(6) txsz)
    return 64 - UInt(txsz);
```

G1.1.4.12 AArch64.NextTableBase

```
// AArch64.NextTableBase()
// =======================
// Extract the address embedded in a table descriptor pointing to the base of
// the next level table of descriptors

bits(52) AArch64.NextTableBase(bits(N) descriptor, TGx tgx)
    bits(52) tablebase = Zeros(52);
    constant AddressSize granulebits = TGxGranuleBits(tgx);
    constant integer tablesize = granulebits;

    case tgx of
        when TGx_4KB  tablebase<47:12> = descriptor<47:12>;
        when TGx_16KB tablebase<47:14> = descriptor<47:14>;
        when TGx_64KB tablebase<47:16> = descriptor<47:16>;

    tablebase = Align(tablebase, 2^tablesize);

    if tgx == TGx_64KB && (AArch64.PAMax() >= 52 ||
            boolean IMPLEMENTATION_DEFINED "descriptor[15:12] for 64KB granule are OA[51:48]") then
        tablebase<51:48> = descriptor<15:12>;

    return tablebase;
```

G1.1.4.13 AArch64.PhysicalAddressSize

```
// AArch64.PhysicalAddressSize()
// =============================
// Retrieve the number of bits bounding the physical address
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
AddressSize AArch64.PhysicalAddressSize(bits(3) encoded_ps, TGx tgx)
    integer ps;
    integer max_ps;

    case encoded_ps of
        when '000'  ps = 32;
        when '001'  ps = 36;
        when '010'  ps = 40;
        when '011'  ps = 42;
        when '100'  ps = 44;
        when '101'  ps = 48;
        when '110'  ps = 52;
        otherwise
            ps = integer IMPLEMENTATION_DEFINED "Reserved Intermediate Physical Address size value";

    if IsFeatureImplemented(FEAT_LPA) && tgx == TGx_64KB then
        max_ps = Min(52, AArch64.PAMax());
    else
        max_ps = Min(48, AArch64.PAMax());

    return Min(ps, max_ps);
```

G1.1.4.14 AArch64.S1LeafBase

```
// AArch64.S1LeafBase()
// ====================
// Extract the address embedded in a block and page descriptor pointing to the
// base of a memory block

bits(52) AArch64.S1LeafBase(bits(N) descriptor, S1TTWParams walkparams, integer level)
    bits(52) leafbase = Zeros(52);

    granulebits  = TGxGranuleBits(walkparams.tgx);
    descsizelog2 = 3;
    constant integer stride = granulebits - descsizelog2;
    constant integer leafsize = granulebits + stride * (FINAL_LEVEL - level);

    leafbase<47:0> = Align(descriptor<47:0>, 2^leafsize);

    if walkparams.tgx == TGx_64KB && (AArch64.PAMax() >= 52 ||
            boolean IMPLEMENTATION_DEFINED "descriptor[15:12] for 64KB granule are OA[51:48]") then
        leafbase<51:48> = descriptor<15:12>;

    return leafbase;
```

G1.1.4.15 AArch64.S1SLTTEntryAddress

```
// AArch64.S1SLTTEntryAddress()
// ============================
// Compute the first stage 1 translation table descriptor address within the
// table pointed to by the base at the start level

FullAddress AArch64.S1SLTTEntryAddress(integer level, S1TTWParams walkparams,
                                       bits(64) ia, FullAddress tablebase)
    // Input Address size
    iasize       = AArch64.IASize(walkparams.txsz);
    granulebits  = TGxGranuleBits(walkparams.tgx);
    constant integer descsizelog2 = 3;
    stride       = granulebits - descsizelog2;
    levels       = FINAL_LEVEL - level;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    bits(52) index;
    constant AddressSize lsb = levels*stride + granulebits;
    constant AddressSize msb = iasize - 1;
    index = ZeroExtend(ia<msb:lsb>:Zeros(descsizelog2), 52);

    FullAddress descaddress;
    descaddress.address = tablebase.address OR index;
    descaddress.paspace = tablebase.paspace;

    return descaddress;
```

G1.1.4.16 AArch64.S1StartLevel

```
// AArch64.S1StartLevel()
// ======================
// Compute the initial lookup level when performing a stage 1 translation
// table walk

integer AArch64.S1StartLevel(S1TTWParams walkparams)
    // Input Address size
    iasize       = AArch64.IASize(walkparams.txsz);
    granulebits  = TGxGranuleBits(walkparams.tgx);
    constant integer descsizelog2 = 3;
    constant integer stride = granulebits - descsizelog2;
    integer s1startlevel = FINAL_LEVEL - (((iasize-1) - granulebits) DIV stride);
    return s1startlevel;
```

G1.1.4.17 AArch64.S1TTBaseAddress

```
// AArch64.S1TTBaseAddress()
// =========================
// Retrieve the PA/IPA pointing to the base of the initial translation table of stage 1

bits(52) AArch64.S1TTBaseAddress(S1TTWParams walkparams, Regime regime, bits(N) ttbr)
    bits(52) tablebase = Zeros(52);

    // Input Address size
    iasize      = AArch64.IASize(walkparams.txsz);
    granulebits = TGxGranuleBits(walkparams.tgx);
    descsizelog2 = 3;
    stride      = granulebits - descsizelog2;
    startlevel  = AArch64.S1StartLevel(walkparams);
    levels      = FINAL_LEVEL - startlevel;

    // Base address is aligned to size of the initial translation table in bytes
    tsize = (iasize - (levels*stride + granulebits)) + descsizelog2;

    if walkparams.tgx == TGx_64KB && walkparams.ps == '110' && (IsFeatureImplemented(FEAT_LPA) ||
            boolean IMPLEMENTATION_DEFINED "BADDR expresses 52 bits for 64KB granule") then
        tsize = Max(tsize, 6);
        tablebase<51:6> = ttbr<5:2>:ttbr<47:6>;
    else
        tablebase<47:1> = ttbr<47:1>;
    tablebase = Align(tablebase, 2^tsize);
    return tablebase;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.1.4.18 AArch64.S1TTEntryAddress

```
// AArch64.S1TTEntryAddress()
// ==========================
// Compute translation table descriptor address within the table pointed to by
// the table base

FullAddress AArch64.S1TTEntryAddress(integer level, S1TTWParams walkparams, bits(64) ia,
                                     FullAddress tablebase, bits(N) descriptor)
    // Input Address size
    iasize      = AArch64.IASize(walkparams.txsz);
    granulebits = TGxGranuleBits(walkparams.tgx);
    constant descsizelog2 = 3;
    stride      = granulebits - descsizelog2;
    levels      = FINAL_LEVEL - level;

    bits(52) index;

    constant AddressSize lsb = levels*stride + granulebits;
    constant AddressSize msb = Min(iasize - 1, (lsb + stride) - 1);
    index = ZeroExtend(ia<msb:lsb>:Zeros(descsizelog2), 52);

    FullAddress descaddress;
    descaddress.address = tablebase.address OR index;
    descaddress.paspace = tablebase.paspace;

    return descaddress;
```

G1.1.4.19 AArch64.AddrTop

```
// AArch64.AddrTop()
// =================
// Get the top bit position of the virtual address.
// Bits above are not accounted as part of the translation process.

AddressSize AArch64.AddrTop(bit tbid, AccessType acctype, bit tbi)
    if tbid == '1' && acctype == AccessType_IFETCH then
        return 63;

    if tbi == '1' then
        return 55;
    else
        return 63;
```

G1.1.4.20 AArch64.ContiguousBitFaults

```
// AArch64.ContiguousBitFaults()
// =============================
// If contiguous bit is set, returns whether the translation size exceeds the
// input address size and if the implementation generates a fault

boolean AArch64.ContiguousBitFaults(bits(6) txsz, TGx tgx, integer level)
    // Input Address size
    iasize = AArch64.IASize(txsz);
    // Translation size
    tsize  = TranslationSize(tgx, level) + ContiguousSize(tgx, level);

    return (tsize > iasize &&
            boolean IMPLEMENTATION_DEFINED "Translation fault on misprogrammed contiguous bit");
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.1.4.21 AArch64.OAOutOfRange

```
// AArch64.OAOutOfRange()
// ======================
// Returns whether output address is expressed in the configured size number of bits

boolean AArch64.OAOutOfRange(bits(52) address, bits(3) ps, TGx tgx)
    // Output Address size
    constant integer oasize = AArch64.PhysicalAddressSize(ps, tgx);

    if oasize < 52 then
        return !IsZero(address<51:oasize>);
    else
        return FALSE;
```

G1.1.4.22 AArch64.S1CheckPermissions

```
// AArch64.S1CheckPermissions()
// ============================
// Checks whether stage 1 access violates permissions of target memory
// and returns a fault record

FaultRecord AArch64.S1CheckPermissions(FaultRecord fault_in, bits(64) va, integer size,
                                       Regime regime, TTWState walkstate, S1TTWParams walkparams,
                                       AccessDescriptor accdesc)
    FaultRecord fault = fault_in;
    constant Permissions permissions = walkstate.permissions;
    S1AccessControls s1perms = AArch64.S1ComputePermissions(regime, walkstate, walkparams, accdesc);

    if accdesc.acctype == AccessType_IFETCH then
        if (walkstate.memattrs.memtype == MemType_Device &&
                ConstrainUnpredictable(Unpredictable_INSTRDEVICE) == Constraint_FAULT) then
            fault.statuscode = Fault_Permission;
        elsif s1perms.x == '0' then
            fault.statuscode = Fault_Permission;
    elsif accdesc.acctype == AccessType_DC then
        if accdesc.cacheop == CacheOp_Invalidate then
            if s1perms.w == '0' then
                fault.statuscode = Fault_Permission;
        // DC from privileged context which clean cannot generate a Permission fault
        elsif accdesc.el == EL0 then
            if s1perms.r == '0' then
                fault.statuscode = Fault_Permission;
    elsif accdesc.acctype == AccessType_IC then
        // IC from privileged context cannot generate Permission fault
        if accdesc.el == EL0 then
            if (s1perms.r == '0' &&
                  boolean IMPLEMENTATION_DEFINED "Permission fault on EL0 IC_IVAU execution") then
                fault.statuscode = Fault_Permission;
    elsif accdesc.read && s1perms.r == '0' then
        fault.statuscode = Fault_Permission;
        fault.write      = FALSE;
    elsif accdesc.write && s1perms.w == '0' then
        fault.statuscode = Fault_Permission;
        fault.write      = TRUE;

    return fault;

FaultRecord AArch64.S1CheckPermissions_PMSA(FaultRecord fault_in, Regime regime,
                                            MemoryAttributes memattrs, Permissions permissions,
                                            AccessDescriptor accdesc)
    bit r, w, x;
    FaultRecord fault = fault_in;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if HasUnprivileged(regime) then
    bit pr, pw;
    bit ur, uw;
    // Apply MPU permissions given Translation Regime serves 2 ELs
    case permissions.ap<2:1> of
        when '00' (pr,pw,ur,uw) = ('1','1','0','0'); // Privileged access
        when '01' (pr,pw,ur,uw) = ('1','1','1','1'); // No effect
        when '10' (pr,pw,ur,uw) = ('1','0','0','0'); // Read-only, privileged access
        when '11' (pr,pw,ur,uw) = ('1','0','1','0'); // Read-only

    if AArch64.IsStage1VMSA(regime) then
        // Apply hierarchical permissions for stage 1 VMSA
        case permissions.ap_table of
            when '00' (pr,pw,ur,uw) = ( pr, pw, ur, uw); // No effect
            when '01' (pr,pw,ur,uw) = ( pr, pw,'0','0'); // Privileged access
            when '10' (pr,pw,ur,uw) = ( pr,'0', ur,'0'); // Read-only
            when '11' (pr,pw,ur,uw) = ( pr,'0','0','0'); // Read-only, privileged access
    // Locations writable by unprivileged cannot be executed by privileged
    // unless SCTLR_ELx.{M,BR} is effectively '01'
    bit px;
    if !AArch64.S1Enabled(regime, accdesc.acctype) && AArch64.S1BREnabled(regime) then
        px = NOT(permissions.pxn);
    else
        px = NOT(permissions.pxn OR uw);
    ux = NOT(permissions.uxn);

    bit pan;
    if (IsFeatureImplemented(FEAT_PAN) && AArch64.S1Enabled(regime, accdesc.acctype) &&
          accdesc.pan) then
        pan = PSTATE.PAN AND (ur OR uw);
    else
        pan = '0';

    pr = pr AND NOT(pan);
    pw = pw AND NOT(pan);
    (r,w,x) = if accdesc.el == EL0 then (ur,uw,ux) else (pr,pw,px);
else
    // Apply MPU permissions given Translation Regime serves 1 EL
    case permissions.ap<2> of
        when '0' (r,w) = ('1','1'); // No effect
        when '1' (r,w) = ('1','0'); // Read-only
    x = NOT(permissions.xn);

// Prevent execution from writable locations if WXN is effectively set
case regime of
    when Regime_EL2  x = x AND NOT(SCTLR_EL2.WXN AND w);
    when Regime_EL10 x = x AND NOT(SCTLR_EL1.WXN AND w);

if accdesc.acctype == AccessType_IFETCH then
    if (ConstrainUnpredictable(Unpredictable_INSTRDEVICE) == Constraint_FAULT &&
            memattrs.memtype == MemType_Device) then
        fault.statuscode = Fault_Permission;

    elsif x == '0' then
        fault.statuscode = Fault_Permission;

elsif accdesc.acctype == AccessType_DC then
    if accdesc.cacheop == CacheOp_Invalidate then
        if w == '0' then
            fault.statuscode = Fault_Permission;

    else
        // DC ops from privileged context which clean cannot generate a Permission fault
        if accdesc.el == EL0 && r == '0' then
            fault.statuscode = Fault_Permission;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif accdesc.acctype == AccessType_IC then
        // IC from privileged context cannot Permission fault
        if (accdesc.el == EL0 && r == '0' &&
                boolean IMPLEMENTATION_DEFINED "Permission fault on EL0 IC_IVAU execution") then
            fault.statuscode = Fault_Permission;

    elsif accdesc.read && r == '0' then
        fault.statuscode = Fault_Permission;
        fault.write      = FALSE;

    elsif accdesc.write && w == '0' then
        fault.statuscode = Fault_Permission;
        fault.write      = TRUE;

    return fault;

FaultRecord AArch64.S2CheckPermissions(FaultRecord fault_in, MemoryAttributes memattrs,
                                       Permissions permissions, AccessDescriptor accdesc)
    // Stage 2 AP follows a similar mapping as stage 1
    // EL1 & EL0 are treated in the same way EL0 is in stage 1
    bit r;
    bit w;
    FaultRecord fault = fault_in;
    case permissions.ap<2:1> of
        when 'x0' (r,w) = ('0','0'); // No access
        when '01' (r,w) = ('1','1'); // No effect
        when '11' (r,w) = ('1','0'); // Read-only
    bit px;
    bit ux;
    case (permissions.xn:permissions.s2xnx) of
        when '00' (px,ux) = ('1','1');
        when '01' (px,ux) = ('0','1');
        when '10' (px,ux) = ('0','0');
        when '11' (px,ux) = ('1','0');

    x = if accdesc.el == EL0 then ux else px;

    if accdesc.acctype == AccessType_TTW then
        if HCR_EL2.PTW == '1' && memattrs.memtype == MemType_Device then
            fault.statuscode = Fault_Permission;

        elsif r == '0' then
            fault.statuscode = Fault_Permission;

        elsif accdesc.write && w == '0' then
            fault.statuscode = Fault_Permission;

    elsif accdesc.acctype == AccessType_IFETCH then
        constraint = ConstrainUnpredictable(Unpredictable_INSTRDEVICE);
        if constraint == Constraint_FAULT && memattrs.memtype == MemType_Device then
            fault.statuscode = Fault_Permission;

        elsif x == '0' then
            fault.statuscode = Fault_Permission;

    elsif accdesc.acctype == AccessType_DC then
        if accdesc.cacheop == CacheOp_Invalidate then
            if w == '0' then
                fault.statuscode = Fault_Permission;

        else
            // DC from privileged context which do no write cannot Permission fault
            if accdesc.el == EL0 && r == '0' then
                fault.statuscode = Fault_Permission;

    elsif accdesc.acctype == AccessType_IC then
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        // IC from privileged context cannot Permission fault
        if (accdesc.el == EL0 && r == '0' &&
                boolean IMPLEMENTATION_DEFINED "Permission fault on EL0 IC_IVAU execution") then
            fault.statuscode = Fault_Permission;
    elsif accdesc.read && r == '0' then
        fault.statuscode = Fault_Permission;
        fault.write      = FALSE;

    elsif accdesc.write && w == '0' then
        fault.statuscode = Fault_Permission;
        fault.write      = TRUE;

    return fault;
```

G1.1.4.23 AArch64.S1ComputePermissions

```
// AArch64.S1ComputePermissions()
// ==============================
// Computes the overall stage 1 permissions

S1AccessControls AArch64.S1ComputePermissions(Regime regime, TTWState walkstate,
                                              S1TTWParams walkparams, AccessDescriptor accdesc)
    constant Permissions permissions = walkstate.permissions;
    S1AccessControls s1perms;

    s1perms = AArch64.S1DirectBasePermissions(regime, walkstate, walkparams, accdesc);

    s1perms.x = s1perms.x AND NOT(s1perms.wxn);

    return s1perms;
```

G1.1.4.24 AArch64.S1DirectBasePermissions

```
// AArch64.S1DirectBasePermissions()
// =================================
// Computes the stage 1 direct base permissions

S1AccessControls AArch64.S1DirectBasePermissions(Regime regime, TTWState walkstate,
                                                 S1TTWParams walkparams, AccessDescriptor accdesc)
    bit  r,  w,  x;
    bit pr, pw, px;
    bit ur, uw, ux;
    Permissions permissions = walkstate.permissions;
    S1AccessControls s1perms;
    // Descriptors marked with DBM set have the effective value of AP[2] cleared.
    // This implies no Permission faults caused by lack of write permissions are
    // reported, and the Dirty bit can be set.
    if permissions.dbm == '1' && walkparams.hd == '1' then
        permissions.ap<2> = '0';

    if HasUnprivileged(regime) then
        // Apply leaf permissions
        case permissions.ap<2:1> of
            when '00' (pr,pw,ur,uw) = ('1','1','0','0'); // Privileged access
            when '01' (pr,pw,ur,uw) = ('1','1','1','1'); // No effect
            when '10' (pr,pw,ur,uw) = ('1','0','0','0'); // Read-only, privileged access
            when '11' (pr,pw,ur,uw) = ('1','0','1','0'); // Read-only

        // Apply hierarchical permissions
        case permissions.ap_table of
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            when '00' (pr,pw,ur,uw) = ( pr, pw, ur, uw); // No effect
            when '01' (pr,pw,ur,uw) = ( pr, pw,'0','0'); // Privileged access
            when '10' (pr,pw,ur,uw) = ( pr,'0', ur,'0'); // Read-only
            when '11' (pr,pw,ur,uw) = ( pr,'0','0','0'); // Read-only, privileged access

        // Locations writable by unprivileged cannot be executed by privileged
        px = NOT(permissions.pxn OR permissions.pxn_table OR uw);
        ux = NOT(permissions.uxn OR permissions.uxn_table);

        if IsFeatureImplemented(FEAT_PAN) && accdesc.pan then
            bit pan;
            pan = PSTATE.PAN AND (ur OR uw);
            pr = pr AND NOT(pan);
            pw = pw AND NOT(pan);

    else
        // Apply leaf permissions
        case permissions.ap<2> of
            when '0' (pr,pw) = ('1','1'); // No effect
            when '1' (pr,pw) = ('1','0'); // Read-only

        // Apply hierarchical permissions
        case permissions.ap_table<1> of
            when '0' (pr,pw) = ( pr, pw); // No effect
            when '1' (pr,pw) = ( pr,'0'); // Read-only

        px = NOT(permissions.xn OR permissions.xn_table);

    (r,w,x) = if accdesc.el == EL0 then (ur,uw,ux) else (pr,pw,px);

    // Compute WXN value
    wxn = walkparams.wxn AND w AND x;

    s1perms.r   = r;
    s1perms.w   = w;
    s1perms.x   = x;
    s1perms.wxn = wxn;

    return s1perms;
```

G1.1.4.25 AArch64.S1HasAlignmentFaultDueToMemType

```
// AArch64.S1HasAlignmentFaultDueToMemType()
// =========================================
// Returns whether stage 1 output fails alignment requirement on data accesses due to memory type

boolean AArch64.S1HasAlignmentFaultDueToMemType(Regime regime, AccessDescriptor accdesc,
                                                boolean aligned,
                                                MemoryAttributes memattrs)

    if accdesc.exclusive || accdesc.atomicop || accdesc.acqsc || accdesc.acqpc || accdesc.relsc then
        if (!aligned && !(IsWBShareable(memattrs) && AArch64.S1DCacheEnabled(regime)) &&
                ConstrainUnpredictableBool(Unpredictable_LSE2_ALIGNMENT_FAULT)) then
            return TRUE;

    if memattrs.memtype != MemType_Device then
        return FALSE;
    elsif accdesc.acctype == AccessType_DCZero then
        return TRUE;
    elsif !aligned then
        return !(boolean IMPLEMENTATION_DEFINED "Device location supports unaligned access");
    else
        return FALSE;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.1.4.26 AArch64.S1OAOutOfRange

```
// AArch64.S1OAOutOfRange()
// ========================
// Returns whether stage 1 output address is expressed in the configured size number of bits

boolean AArch64.S1OAOutOfRange(bits(52) address, S1TTWParams walkparams)
    return AArch64.OAOutOfRange(address, walkparams.ps, walkparams.tgx);
```

G1.1.4.27 AArch64.S1TxSZFaults

```
// AArch64.S1TxSZFaults()
// ======================
// Detect whether configuration of stage 1 TxSZ field generates a fault

boolean AArch64.S1TxSZFaults(Regime regime, S1TTWParams walkparams)
    mintxsz = AArch64.S1MinTxSZ(regime, walkparams);
    maxtxsz = AArch64.MaxTxSZ(walkparams.tgx);

    if UInt(walkparams.txsz) < mintxsz then
        return (IsFeatureImplemented(FEAT_LVA) ||
                boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value below minimum");
    if UInt(walkparams.txsz) > maxtxsz then
        return boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value above maximum";

    return FALSE;
```

G1.1.4.28 AArch64.S2HasAlignmentFaultDueToMemType

```
// AArch64.S2HasAlignmentFaultDueToMemType()
// =========================================
// Returns whether stage 2 output fails alignment requirement on data accesses due to memory type

boolean AArch64.S2HasAlignmentFaultDueToMemType(AccessDescriptor accdesc, boolean aligned,
                                                MemoryAttributes memattrs)

    if accdesc.exclusive || accdesc.atomicop || accdesc.acqsc || accdesc.acqpc || accdesc.relsc then
        if (!aligned && !(IsWBShareable(memattrs) && S2DCacheEnabled()) &&
                ConstrainUnpredictableBool(Unpredictable_LSE2_ALIGNMENT_FAULT)) then
            return TRUE;

    if memattrs.memtype != MemType_Device then
        return FALSE;
    elsif accdesc.acctype == AccessType_DCZero then
        return TRUE;
    elsif !aligned then
        return !(boolean IMPLEMENTATION_DEFINED "Device location supports unaligned access");
    else
        return FALSE;
```

G1.1.4.29 AArch64.VAIsOutOfRange

```
// AArch64.VAIsOutOfRange()
// ========================
// Check bits not resolved by translation are identical and of accepted value
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
boolean AArch64.VAIsOutOfRange(bits(64) va_in, AccessType acctype,
                               Regime regime, S1TTWParams walkparams)
    bits(64) va = va_in;

    constant AddressSize addrtop = AArch64.AddrTop(walkparams.tbid, acctype, walkparams.tbi);

    // Input Address size
    constant integer iasize = AArch64.IASize(walkparams.txsz);

    if HasUnprivileged(regime) then
        if AArch64.GetVARange(va) == VARange_LOWER then
            return !IsZero(va<addrtop:iasize>);
        else
            return !IsOnes(va<addrtop:iasize>);
    else
        return !IsZero(va<addrtop:iasize>);
```

G1.1.4.30 AArch64.S2ApplyFWBMemAttrs

```
// AArch64.S2ApplyFWBMemAttrs()
// ============================
// Apply stage 2 transformation on stage 1 memory attributes using FWB mapping.

MemoryAttributes AArch64.S2ApplyFWBMemAttrs(MemoryAttributes s1_mematTrs,
                                            bits(8) s2_attr, bits(2) s2_sh)
    MemoryAttributes mematTrs;

    // If inner and outer memory attribute encoding are not identical,
    // the combined memory atributes are UNKNOWN
    if s2_attr<7:4> != s2_attr<3:0> then
        mematTrs = MemoryAttributes UNKNOWN;

    elsif s2_attr<7:6> == '11' then // Force writeback
        mematTrs.memtype = MemType_Normal;

        mematTrs.inner.attrs = MemAttr_WB;
        if (s1_mematTrs.memtype == MemType_Normal &&
                s1_mematTrs.inner.attrs != MemAttr_NC) then
            mematTrs.inner.hints     = s1_mematTrs.inner.hints;
            mematTrs.inner.transient = s1_mematTrs.inner.transient;
        else
            mematTrs.inner.hints     = MemHint_RWA;
            mematTrs.inner.transient = FALSE;

        mematTrs.outer.attrs = MemAttr_WB;
        if (s1_mematTrs.memtype == MemType_Normal &&
                s1_mematTrs.outer.attrs != MemAttr_NC) then
            mematTrs.outer.hints     = s1_mematTrs.outer.hints;
            mematTrs.outer.transient = s1_mematTrs.outer.transient;
        else
            mematTrs.outer.hints     = MemHint_RWA;
            mematTrs.outer.transient = FALSE;

        s2_shareability = DecodeShareability(s2_sh);
        mematTrs.shareability = S2CombineS1Shareability(s1_mematTrs.shareability, s2_shareability);
    else
        // Follow memory decoding & combining rules as if FWB is inactive
        // V8R stage 2 memory attributes are decoded in the same fashion as stage 1
        // for the EL2 regime. The only difference is allocation hints are ignored
        s1aarch64   = TRUE;
        s2_mematTrs = S1DecodeMemAttrs(s2_attr, s2_sh, s1aarch64);
        s2aarch64   = TRUE;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        memattrs    = S2CombineS1MemAttrs(s1_memattrs, s2_memattrs, s2aarch64);

    return memattrs;
```

G1.1.4.31 AArch64.GetS1TLBContext

```
// AArch64.GetS1TLBContext()
// =========================
// Gather translation context for accesses with VA to match against TLB entries

TLBContext AArch64.GetS1TLBContext(Regime regime, SecurityState ss, bits(64) va, TGx tg)
    TLBContext tlbcontext;

    case regime of
        when Regime_EL10 tlbcontext = AArch64.TLBContextEL10(ss, va, tg);
        otherwise
            Unreachable();

    tlbcontext.includes_s1  = TRUE;
    // The following may be amended for EL1&0 Regime if caching of stage 2 is successful
    tlbcontext.includes_s2  = FALSE;
    tlbcontext.use_vmid     = UseVMID(regime);
    return tlbcontext;
```

G1.1.4.32 AArch64.TLBContextEL10

```
// AArch64.TLBContextEL10()
// ========================
// Gather translation context for accesses under EL10 regime to match against TLB entries

TLBContext AArch64.TLBContextEL10(SecurityState ss, bits(64) va, TGx tg)
    TLBContext tlbcontext;

    tlbcontext.ss     = ss;
    tlbcontext.regime = Regime_EL10;
    tlbcontext.vmid   = VMID[];

    if TCR_EL1.A1 == '0' || VTCR_EL2.MSA == '0' then
        tlbcontext.asid = TTBR0_EL1.ASID;
    else
        tlbcontext.asid = TTBR1_EL1.ASID;

    if TCR_EL1.AS == '0' then
        tlbcontext.asid<15:8> = Zeros(8);
    tlbcontext.tg     = tg;
    tlbcontext.ia     = va;

    if IsFeatureImplemented(FEAT_TTCNP) && VTCR_EL2.MSA == '1' then
        if AArch64.GetVARange(va) == VARange_LOWER then
            tlbcontext.cnp = TTBR0_EL1.CnP;
        else
            tlbcontext.cnp = TTBR1_EL1.CnP;
    else
        tlbcontext.cnp = '0';

    return tlbcontext;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.1.4.33 AArch64.MemSwapTableDesc

```
// AArch64.MemSwapTableDesc()
// ==========================
// Perform HW update of table descriptor as an atomic operation

(FaultRecord, bits(N)) AArch64.MemSwapTableDesc(FaultRecord fault_in, bits(N) prev_desc,
                                                bits(N) new_desc, bit ee,
                                                AccessDescriptor  descaccess,
                                                AddressDescriptor descpaddr, integer N)
    assert descaccess.acctype == AccessType_TTW;
    FaultRecord fault = fault_in;
    boolean iswrite;

    // All observers in the shareability domain observe the
    // following memory read and write accesses atomically.
    bits(N) mem_desc;
    PhysMemRetStatus memstatus;
    (memstatus, mem_desc) = PhysMemRead(descpaddr, N DIV 8, descaccess);

    if ee == '1' then
        mem_desc = BigEndianReverse(mem_desc);

    if IsFault(memstatus) then
        iswrite = FALSE;
        fault = HandleExternalTTWAbort(memstatus, iswrite, descpaddr, descaccess, N DIV 8, fault);
        if IsFault(fault.statuscode) then
            return (fault, bits(N) UNKNOWN);

    if mem_desc == prev_desc  then
        ordered_new_desc = if ee == '1' then BigEndianReverse(new_desc) else new_desc;
        memstatus = PhysMemWrite(descpaddr, N DIV 8, descaccess, ordered_new_desc);

        if IsFault(memstatus) then
            iswrite = TRUE;
            fault = HandleExternalTTWAbort(memstatus, iswrite, descpaddr, descaccess, N DIV 8,
                                           fault);

            if IsFault(fault.statuscode) then
                return (fault, bits(N) UNKNOWN);

        // Reflect what is now in memory (in little endian format)
        mem_desc = new_desc;

    return (fault, mem_desc);
```

G1.1.4.34 AArch64.S1DisabledOutput

```
// AArch64.S1DisabledOutput()
// ==========================
// Map the VA to IPA/PA and assign default memory attributes

(FaultRecord, AddressDescriptor) AArch64.S1DisabledOutput(FaultRecord fault_in, Regime regime,
                                                          bits(64) va, AccessDescriptor accdesc,
                                                          boolean aligned)
    bit tbi;
    bit tbid;
    FaultRecord fault = fault_in;
    case regime of
        when Regime_EL2
            tbi  = TCR_EL2.TBI;
            tbid = TCR_EL2.TBID;
        when Regime_EL10
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if AArch64.IsStage1VMSA(regime) then
            if AArch64.GetVARange(va) == VARange_LOWER then
                tbi  = TCR_EL1.TBI0;
                tbid = TCR_EL1.TBID0;
            else
                tbi  = TCR_EL1.TBI1;
                tbid = TCR_EL1.TBID1;
        else
            tbi  = TCR_EL1.TBI0;
            tbid = TCR_EL1.TBID0;

    // Output Address
    FullAddress oa;
    oa.address = va<51:0>;
    if accdesc.ss == SS_Secure then
        oa.paspace = PAS_Secure;
    else
        oa.paspace = PAS_NonSecure;

    MemoryAttributes memattrs;
    if regime == Regime_EL10 && EL2Enabled() && HCR_EL2.DC == '1' then
        MemAttrHints default_cacheability;
        default_cacheability.attrs     = MemAttr_WB;
        default_cacheability.hints     = MemHint_RWA;
        default_cacheability.transient = FALSE;

        memattrs.memtype      = MemType_Normal;
        memattrs.outer        = default_cacheability;
        memattrs.inner        = default_cacheability;
        memattrs.shareability = Shareability_NSH;
    elsif accdesc.acctype == AccessType_IFETCH then
        MemAttrHints i_cache_attr;
        if AArch64.S1ICacheEnabled(regime) then
            i_cache_attr.attrs     = MemAttr_WT;
            i_cache_attr.hints     = MemHint_RA;
            i_cache_attr.transient = FALSE;
        else
            i_cache_attr.attrs     = MemAttr_NC;

        memattrs.memtype      = MemType_Normal;
        memattrs.outer        = i_cache_attr;
        memattrs.inner        = i_cache_attr;
        memattrs.shareability = Shareability_OSH;
    else
        memattrs.memtype      = MemType_Device;
        memattrs.device       = DeviceType_nGnRnE;
        memattrs.shareability = Shareability_OSH;

    fault.level = 0;
    constant AddressSize addrtop = AArch64.AddrTop(tbid, accdesc.acctype, tbi);
    constant AddressSize pamax = AArch64.PAMax();
    if !IsZero(va<addrtop:pamax>) then
        fault.statuscode = Fault_AddressSize;
    elsif AArch64.S1HasAlignmentFaultDueToMemType(regime, accdesc, aligned, memattrs) then
        fault.statuscode = Fault_Alignment;

    if fault.statuscode != Fault_None then
        return (fault, AddressDescriptor UNKNOWN);
    else
        ipa = CreateAddressDescriptor(va, oa, memattrs, accdesc);
        return (fault, ipa);
```

G1.1.4.35 AArch64.S1Translate

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// AArch64.S1Translate()
// =====================
// Translate VA to IPA/PA depending on the regime

(FaultRecord, AddressDescriptor) AArch64.S1Translate(FaultRecord fault_in, Regime regime,
                                                     bits(64) va, integer size, boolean aligned,
                                                     AccessDescriptor accdesc)
    FaultRecord fault = fault_in;
    // Prepare fault fields in case a fault is detected
    fault.secondstage = FALSE;
    fault.s2fs1walk   = FALSE;

    if !AArch64.S1Enabled(regime, accdesc.acctype) then
        return AArch64.S1DisabledOutput(fault, regime, va, accdesc, aligned);

    walkparams = AArch64.GetS1TTWParams(regime, accdesc.el, accdesc.ss, va);
    constant integer s1mintxsz = AArch64.S1MinTxSZ(regime, walkparams);
    constant integer s1maxtxsz = AArch64.MaxTxSZ(walkparams.tgx);
    if AArch64.S1TxSZFaults(regime, walkparams) then
        fault.statuscode = Fault_Translation;
        fault.level      = 0;
        return (fault, AddressDescriptor UNKNOWN);
    elsif UInt(walkparams.txsz) < s1mintxsz then
        walkparams.txsz = s1mintxsz<5:0>;
    elsif UInt(walkparams.txsz) > s1maxtxsz then
        walkparams.txsz = s1maxtxsz<5:0>;

    if AArch64.VAIsOutOfRange(va, accdesc.acctype, regime, walkparams) then
        fault.statuscode = Fault_Translation;
        fault.level      = 0;
        return (fault, AddressDescriptor UNKNOWN);

    if accdesc.el == EL0 && walkparams.e0pd == '1' then
        fault.statuscode = Fault_Translation;
        fault.level      = 0;
        return (fault, AddressDescriptor UNKNOWN);

    AddressDescriptor descipaddr;
    TTWState walkstate;
    bits(64) descriptor;
    (fault, descipaddr, walkstate, descriptor) = AArch64.S1Walk(fault, walkparams, va, regime,
                                                                accdesc, 64);
    if fault.statuscode != Fault_None then
        return (fault, AddressDescriptor UNKNOWN);
    if AArch64.S1HasAlignmentFaultDueToMemType(regime, accdesc, aligned,
                                               walkstate.mematttrs) then
        fault.statuscode = Fault_Alignment;

    constant FaultRecord fault_perm = AArch64.S1CheckPermissions(fault, va, size, regime,
                                                                 walkstate, walkparams, accdesc);

    bits(64) mem_desc;
    bits(64) new_desc = descriptor;

    if AArch64.SetAccessFlag(walkparams.ha, accdesc, fault) then
        // Set descriptor AF bit
        new_desc<10> = '1';

    // If HW update of dirty bit is enabled, the walk state permissions
    // will already reflect a configuration permitting writes.
    // The update of the descriptor occurs only if the descriptor bits in
    // memory do not reflect that and the access instigates a write.

    if AArch64.SetDirtyState(walkparams.hd, descriptor<51>, accdesc, fault, fault_perm) then
        // Clear descriptor AP[2]/nDirty bit permitting stage 1 writes
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        new_desc<7> = '0';

    if fault.statuscode == Fault_None && fault_perm.statuscode != Fault_None then
        fault = fault_perm;

    // Either the access flag was clear or AP[2]/nDirty was set
    if new_desc != descriptor then
        if (!IsWBShareable(descipaddr.memattrs) &&
              ConstrainUnpredictableBool(Unpredictable_Unsupported_Atomic_HW_Update)) then
            fault.statuscode = Fault_HWUpdateAccessFlag;
            return (fault, AddressDescriptor UNKNOWN);

        AddressDescriptor descpaddr;
        descaccess = CreateAccDescTTEUpdate(accdesc);

        FaultRecord s2fault;
        s2aligned = TRUE;
        (s2fault, descpaddr) = AArch64.S2Validate(fault, descipaddr, s2aligned, descaccess);

        if s2fault.statuscode != Fault_None then
            return (s2fault, AddressDescriptor UNKNOWN);

        (fault, mem_desc) = AArch64.MemSwapTableDesc(fault, descriptor, new_desc, walkparams.ee,
                                                     descaccess, descpaddr, 64);

        if fault.statuscode != Fault_None then
            if (accdesc.acctype == AccessType_AT &&
                    !(boolean IMPLEMENTATION_DEFINED "AT reports the HW update fault")) then
                // Mask the fault
                fault.statuscode = Fault_None;
            else
                return (fault, AddressDescriptor UNKNOWN);
        elsif new_desc != descriptor && mem_desc != new_desc then
            // HW update of Dirty state or AF was not successful due to the descriptor being updated
            // not matching the descriptor used for translation. Due to this, the walk is restarted.
            return AArch64.S1Translate(fault_in, regime, va, size, aligned, accdesc);

    if fault.statuscode != Fault_None then
        return (fault, AddressDescriptor UNKNOWN);

    // Output Address
    oa = StageOA(va, walkparams.tgx, walkstate);
    MemoryAttributes mematTrs;
    if AArch64.S1TreatAsNormalNC(walkstate, regime, accdesc) then
       // Treat memory attributes as Normal Non-Cacheable
        mematTrs = NormalNCMemAttr();

    else
        mematTrs = walkstate.mematTrs;

    // Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
    // to be either effective value or descriptor value
    if (regime == Regime_EL10 && EL2Enabled() && HCR_EL2.VM == '1' &&
            !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1")) then
        mematTrs.shareability = walkstate.mematTrs.shareability;
    else
        mematTrs.shareability = EffectiveShareability(mematTrs);

    ipa = CreateAddressDescriptor(va, oa, mematTrs, accdesc);

    if (accdesc.atomicop && !IsWBShareable(mematTrs) &&
          ConstrainUnpredictableBool(Unpredictable_Atomic_MMU_IMPDEF_FAULT)) then
        fault.statuscode = Fault_Exclusive;
        return (fault, ipa);

    return (fault, ipa);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.1.4.36 AArch64.S1TreatAsNormalNC

```
// AArch64.S1TreatAsNormalNC()
// ===========================
// Returns TRUE if stage 1 memory attributes should be treated as Normal Non-Cacheable

boolean AArch64.S1TreatAsNormalNC(TTWState walkstate, Regime regime, AccessDescriptor accdesc)
    return ((accdesc.acctype == AccessType_IFETCH &&
               (walkstate.mematttrs.memtype == MemType_Device || !AArch64.S1ICacheEnabled(regime))) ||
            (accdesc.acctype != AccessType_IFETCH &&
               !AArch64.S1DCacheEnabled(regime) && walkstate.mematttrs.memtype == MemType_Normal));
```

G1.1.4.37 AArch64.SetAccessFlag

```
// AArch64.SetAccessFlag()
// =======================
// Determine whether the access flag could be set by HW given the fault status

boolean AArch64.SetAccessFlag(bit ha, AccessDescriptor accdesc, FaultRecord fault)
    if ha == '0' || !AArch64.SettingAccessFlagPermitted(fault) then
        return FALSE;
    elsif accdesc.acctype == AccessType_AT then
        return boolean IMPLEMENTATION_DEFINED "AT updates AF";
    elsif accdesc.acctype IN {AccessType_DC, AccessType_IC} then
        return boolean IMPLEMENTATION_DEFINED "Generate access flag fault on IC/DC operations";
    else
        // Set descriptor AF bit
        return TRUE;
```

G1.1.4.38 AArch64.SetDirtyState

```
// AArch64.SetDirtyState()
// =======================
// Determine whether dirty state is required to be updated by HW given the fault status

boolean AArch64.SetDirtyState(bits(1) hd, bits(1) dbm, AccessDescriptor accdesc, FaultRecord fault,
                              FaultRecord fault_perm)
    if hd == '0' then
        return FALSE;
    elsif !AArch64.SettingDirtyStatePermitted(fault, fault_perm) then
        return FALSE;
    elsif accdesc.acctype IN {AccessType_AT, AccessType_IC, AccessType_DC} then
        return FALSE;
    elsif !accdesc.write then
        return FALSE;
    else
        return dbm == '1';
```

G1.1.4.39 AArch64.SettingAccessFlagPermitted

```
// AArch64.SettingAccessFlagPermitted()
// ====================================
// Determine whether the access flag could be set by HW given the fault status

boolean AArch64.SettingAccessFlagPermitted(FaultRecord fault)
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if fault.statuscode == Fault_None then
        return TRUE;
    elsif fault.statuscode IN {Fault_Alignment, Fault_Permission} then
        return ConstrainUnpredictableBool(Unpredictable_AFUPDATE);
    else
        return FALSE;
```

G1.1.4.40 AArch64.SettingDirtyStatePermitted

```
// AArch64.SettingDirtyStatePermitted()
// ====================================
// Determine whether the dirty state could be set by HW given the fault status

boolean AArch64.SettingDirtyStatePermitted(FaultRecord fault, FaultRecord fault_perm)
    if fault_perm.statuscode != Fault_None then
        return FALSE;
    elsif fault.statuscode == Fault_None then
        return TRUE;
    elsif fault.statuscode == Fault_Alignment then
        return ConstrainUnpredictableBool(Unpredictable_DBUPDATE);
    else
        return FALSE;
```

G1.1.4.41 AArch64.TranslateAddress

```
// AArch64.TranslateAddress()
// ==========================
// Main entry point for translating an address

AddressDescriptor AArch64.TranslateAddress(bits(64) va, AccessDescriptor accdesc,
                                           boolean aligned, integer size)
    AddressDescriptor result = AArch64.FullValidate(va, size, accdesc, aligned);

    if !IsFault(result) && accdesc.acctype != AccessType_IFETCH then
        // For an instruction fetch, CheckDebug will be called
        // after the instruction is read from memory
        result.fault = AArch64.CheckDebug(va, accdesc, size);

    // Update virtual address for abort functions
    result.vaddress = ZeroExtend(va, 64);

    return result;
```

G1.1.4.42 AArch64.BlockDescSupported

```
// AArch64.BlockDescSupported()
// ============================
// Determine whether a block descriptor is valid for the given granule size
// and level

boolean AArch64.BlockDescSupported(TGx tgx, integer level)
    case tgx of
        when TGx_4KB  return (level == 1 || level == 2);
        when TGx_16KB return level == 2;
        when TGx_64KB return ((level == 1 && AArch64.PAMax() == 52) ||
                              level == 2);
    return FALSE;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.1.4.43 AArch64.ContiguousBit

```
// AArch64.ContiguousBit()
// =======================
// Get the value of the contiguous bit

bit AArch64.ContiguousBit(TGx tgx, integer level, bits(N) descriptor)
    if boolean IMPLEMENTATION_DEFINED "Treat Contiguous bit as 0" then
        return '0';

    // When using TGx 64KB then the Contiguous bit is RES0 for Block
    // descriptors at level 1. For VMSAv8-64, level 1 Block descriptors
    // can exist only if FEAT_LPA is implemented.

    if tgx == TGx_64KB && level == 1 then
        return '0'; // RES0

    return descriptor<52>;
```

G1.1.4.44 AArch64.DecodeDescriptorType

```
// AArch64.DecodeDescriptorType()
// ==============================
// Determine whether the descriptor is a page, block or table

DescriptorType AArch64.DecodeDescriptorType(bits(N) descriptor,
                                            TGx tgx, integer level)
    if descriptor<0> == '0' then
        return DescriptorType_Invalid;
    else
        if descriptor<1> == '1' then
            if level == FINAL_LEVEL then
                return DescriptorType_Leaf;
            else
                return DescriptorType_Table;
        elsif descriptor<1> == '0' then
            if AArch64.BlockDescSupported(tgx, level) then
                return DescriptorType_Leaf;
            else
                return DescriptorType_Invalid;
        Unreachable();
```

G1.1.4.45 AArch64.S1ApplyOutputPerms

```
// AArch64.S1ApplyOutputPerms()
// ============================
// Apply output permissions encoded in stage 1 page/block descriptors

Permissions AArch64.S1ApplyOutputPerms(Permissions permissions_in, bits(N) descriptor,
                                       Regime regime, S1TTWParams walkparams)
    Permissions permissions = permissions_in;

    if HasUnprivileged(regime) then
        permissions.ap<2:1> = descriptor<7:6>;
        permissions.uxn     = descriptor<54>;
        permissions.pxn     = descriptor<53>;
    else
        permissions.ap<2:1> = descriptor<7>:'1';
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        permissions.xn      = descriptor<54>;
    permissions.dbm = descriptor<51>;

    return permissions;
```

G1.1.4.46 AArch64.S1ApplyTablePerms

```
// AArch64.S1ApplyTablePerms()
// ===========================
// Apply hierarchical permissions encoded in stage 1 table descriptors

Permissions AArch64.S1ApplyTablePerms(Permissions permissions_in, bits(64) descriptor,
                                      Regime regime, S1TTWParams walkparams)
    Permissions permissions = permissions_in;
    bits(2) ap_table;
    bit pxn_table;
    bit uxn_table;
    bit xn_table;
    if HasUnprivileged(regime) then
        ap_table  = descriptor<62:61>;
        uxn_table = descriptor<60>;
        pxn_table = descriptor<59>;
        permissions.ap_table  = permissions.ap_table  OR ap_table;
        permissions.uxn_table = permissions.uxn_table OR uxn_table;
        permissions.pxn_table = permissions.pxn_table OR pxn_table;
    else
        ap_table = descriptor<62>:'0';
        xn_table = descriptor<60>;
        permissions.ap_table = permissions.ap_table OR ap_table;
        permissions.xn_table = permissions.xn_table OR xn_table;

    return permissions;
```

G1.1.4.47 AArch64.nTFaults

```
// AArch64.nTFaults()
// ==================
// Identify whether the nT bit in a block or table descriptor is effectively set
// causing a translation fault

boolean AArch64.nTFaults(bits(N) descriptor)
    if !IsFeatureImplemented(FEAT_BBML1) then
        return FALSE;
    constant bit nT = descriptor<16>;

    return nT == '1' && boolean IMPLEMENTATION_DEFINED "nT bit causes Translation Fault";
```

G1.1.4.48 AArch64.S1InitialTTWState

```
// AArch64.S1InitialTTWState()
// ===========================
// Set properties of first access to translation tables in stage 1

TTWState AArch64.S1InitialTTWState(S1TTWParams walkparams, bits(64) va, Regime regime,
                                   SecurityState ss)
    TTWState    walkstate;
    FullAddress tablebase;
    Permissions permissions;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    bits(64)    ttbr;

    ttbr        = AArch64.S1TTBR(regime, va);
    tablebase.paspace = PAS_Secure;

    tablebase.address = AArch64.S1TTBaseAddress(walkparams, regime, ttbr);

    permissions.ap_table = '00';
    if HasUnprivileged(regime) then
        permissions.uxn_table = '0';
        permissions.pxn_table = '0';
    else
        permissions.xn_table  = '0';

    walkstate.baseaddress = tablebase;
    walkstate.level       = AArch64.S1StartLevel(walkparams);
    walkstate.istable     = TRUE;
    // In regimes that support global and non-global translations, translation
    // table entries from lookup levels other than the final level of lookup
    // are treated as being non-global
    walkstate.nG          = if HasUnprivileged(regime) then '1' else '0';
    walkstate.memattrs    = WalkMemAttrs(walkparams.sh, walkparams.irgn, walkparams.orgn);
    walkstate.permissions = permissions;

    return walkstate;
```

G1.1.4.49 AArch64.S1NextWalkStateLeaf

```
// AArch64.S1NextWalkStateLeaf()
// =============================
// Decode stage 1 page or block descriptor as output to this stage of translation

TTWState AArch64.S1NextWalkStateLeaf(TTWState currentstate, Regime regime,
                                     AccessDescriptor accdesc, S1TTWParams walkparams,
                                     bits(N) descriptor)
    TTWState    nextstate;
    FullAddress baseaddress;
    baseaddress.address = AArch64.S1LeafBase(descriptor, walkparams, currentstate.level);

    if currentstate.baseaddress.paspace == PAS_Secure then
        // Determine PA space of the block from NS bit
        constant bit ns = descriptor<5>;
        baseaddress.paspace = if ns == '0' then PAS_Secure else PAS_NonSecure;
    else
        baseaddress.paspace = PAS_NonSecure;

    nextstate.istable     = FALSE;
    nextstate.level       = currentstate.level;
    nextstate.baseaddress = baseaddress;

    bits(3) attrindx;
    attrindx = descriptor<4:2>;

    bits(2) sh;
    sh = descriptor<9:8>;
    attr = AArch64.MAIRAttr(UInt(attrindx), walkparams.mair);
    s1aarch64 = TRUE;

    nextstate.memattrs = S1DecodeMemAttrs(attr, sh, s1aarch64, walkparams, accdesc.acctype);
    nextstate.permissions = AArch64.S1ApplyOutputPerms(currentstate.permissions,
                                                       descriptor, regime, walkparams);
    nextstate.contiguous  = AArch64.ContiguousBit(walkparams.tgx,
                                                  currentstate.level, descriptor);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if !HasUnprivileged(regime) then
        nextstate.nG = '0';
    elsif accdesc.ss == SS_Secure && currentstate.baseaddress.paspace == PAS_NonSecure then
        // In Secure state, a translation must be treated as non-global,
        // regardless of the value of the nG bit,
        // if NSTable is set to 1 at any level of the translation table walk
        nextstate.nG = '1';
    else
        nextstate.nG = descriptor<11>;

    return nextstate;
```

G1.1.4.50 AArch64.S1NextWalkStateTable

```
// AArch64.S1NextWalkStateTable()
// ==============================
// Decode stage 1 table descriptor to transition to the next level

TTWState AArch64.S1NextWalkStateTable(TTWState currentstate, Regime regime,
                                      S1TTWParams walkparams, bits(N) descriptor)
    TTWState    nextstate;
    FullAddress tablebase;
    tablebase.address = AArch64.NextTableBase(descriptor, walkparams.tgx);

    if currentstate.baseaddress.paspace == PAS_Secure then
        // Determine PA space of the next table from NSTable bit
        bit nstable;
        nstable = descriptor<63>;
        tablebase.paspace = if nstable == '0' then PAS_Secure else PAS_NonSecure;
    else
        // Otherwise bit 63 is RES0 and there is no NSTable bit
        tablebase.paspace = currentstate.baseaddress.paspace;

    nextstate.istable     = TRUE;
    nextstate.nG          = currentstate.nG;
    nextstate.level       = currentstate.level + 1;
    nextstate.baseaddress = tablebase;
    nextstate.memattrs    = currentstate.memattrs;
    if walkparams.hpd == '0' then
        nextstate.permissions = AArch64.S1ApplyTablePerms(currentstate.permissions,
                                                          descriptor<63:0>, regime, walkparams);
    else
        nextstate.permissions = currentstate.permissions;

    return nextstate;
```

G1.1.4.51 AArch64.S1Walk

```
// AArch64.S1Walk()
// ================
// Traverse stage 1 translation tables obtaining the final descriptor
// as well as the address leading to that descriptor

(FaultRecord, AddressDescriptor, TTWState, bits(N)) AArch64.S1Walk(FaultRecord fault_in,
                                                                  S1TTWParams walkparams,
                                                                  bits(64) va, Regime regime,
                                                                  AccessDescriptor accdesc,
                                                                  integer N)
    FaultRecord fault = fault_in;
    boolean aligned;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if HasUnprivileged(regime) && AArch64.S1EPD(regime, va) == '1' then
    fault.statuscode = Fault_Translation;
    fault.level      = 0;
    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(N) UNKNOWN);

walkstate = AArch64.S1InitialTTWState(walkparams, va, regime, accdesc.ss);
constant integer startlevel = walkstate.level;

bits(N) descriptor;
AddressDescriptor walkaddress;
walkaddress.vaddress = va;

if !AArch64.S1DCacheEnabled(regime) then
    walkaddress.memattrs = NormalNCMemAttr();
else
    walkaddress.memattrs = walkstate.memattrs;

// Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
// to be either effective value or descriptor value
if (regime == Regime_EL10 && EL2Enabled() && HCR_EL2.VM == '1' &&
        !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1")) then
    walkaddress.memattrs.shareability = walkstate.memattrs.shareability;
else
    walkaddress.memattrs.shareability = EffectiveShareability(walkaddress.memattrs);

DescriptorType desctype;
FullAddress descaddress = AArch64.S1SLTTEntryAddress(walkstate.level, walkparams, va,
                                                     walkstate.baseaddress);

// Detect Address Size Fault by Descriptor Address
if AArch64.S1OAOutOfRange(descaddress.address, walkparams) then
    fault.statuscode = Fault_AddressSize;
    fault.level      = 0;
    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(N) UNKNOWN);

repeat
    fault.level = walkstate.level;
    walkaddress.paddress = descaddress;

    constant AccessDescriptor walkaccess = CreateAccDescS1TTW(accdesc);
    aligned = TRUE;
    (s2fault, s2walkaddress) = AArch64.S2Validate(fault, walkaddress, aligned, walkaccess);

    if s2fault.statuscode != Fault_None then
        return (s2fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN,
                bits(N) UNKNOWN);

    (fault, descriptor) = FetchDescriptor(walkparams.ee, s2walkaddress, walkaccess, fault, N);

    if fault.statuscode != Fault_None then
        return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN,
                bits(N) UNKNOWN);

    desctype = AArch64.DecodeDescriptorType(descriptor, walkparams.tgx, walkstate.level);

    case desctype of
        when DescriptorType_Table
            walkstate = AArch64.S1NextWalkStateTable(walkstate,
                                                     regime, walkparams, descriptor);
            descaddress = AArch64.S1TTEntryAddress(walkstate.level, walkparams, va,
                                                   walkstate.baseaddress, descriptor);

            // Detect Address Size Fault by Descriptor Address
            if AArch64.S1OAOutOfRange(descaddress.address, walkparams) then
                fault.statuscode = Fault_AddressSize;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(N) UNKNOWN);

            when DescriptorType_Leaf
                walkstate = AArch64.S1NextWalkStateLeaf(walkstate, regime,
                                                        accdesc, walkparams, descriptor);

            when DescriptorType_Invalid
                fault.statuscode = Fault_Translation;
                return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN,
                        bits(N) UNKNOWN);

            otherwise
                Unreachable();
    until desctype == DescriptorType_Leaf;

    constant FullAddress oa = StageOA(va, walkparams.tgx, walkstate);
    if (walkstate.contiguous == '1' &&
        AArch64.ContiguousBitFaults(walkparams.txsz, walkparams.tgx,
                                    walkstate.level)) then
        fault.statuscode = Fault_Translation;
    elsif walkstate.level < FINAL_LEVEL && AArch64.nTFaults(descriptor) then
        fault.statuscode = Fault_Translation;
    // Detect Address Size Fault by final output
    elsif AArch64.S1OAOutOfRange(oa.address, walkparams) then
        fault.statuscode = Fault_AddressSize;
    // Check descriptor AF bit
    elsif (descriptor<10> == '0' && walkparams.ha == '0' &&
           (!accdesc.acctype IN {AccessType_DC, AccessType_IC} ||
            boolean IMPLEMENTATION_DEFINED "Generate access flag fault on IC/DC operations")) then
        fault.statuscode = Fault_AccessFlag;

    if fault.statuscode != Fault_None then
        return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(N) UNKNOWN);

    return (fault, walkaddress, walkstate, descriptor);
```

G1.1.4.52 AArch64.GetS1TTWParams

```
// AArch64.GetS1TTWParams()
// ========================
// Returns stage 1 translation table walk parameters from respective controlling
// System registers.

S1TTWParams AArch64.GetS1TTWParams(Regime regime, bits(2) el, SecurityState ss, bits(64) va)
    S1TTWParams walkparams;

    varange = AArch64.GetVARange(va);

    case regime of
        when Regime_EL10 walkparams = AArch64.S1TTWParamsEL10(el, varange);

    return walkparams;
```

G1.1.4.53 AArch64.GetVARange

```
// AArch64.GetVARange()
// ====================
// Determines if the VA that is to be translated lies in LOWER or UPPER address range.

VARange AArch64.GetVARange(bits(64) va)
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if va<55> == '0' then
        return VARange_LOWER;
    else
        return VARange_UPPER;
```

G1.1.4.54 AArch64.HaveS1TG

```
// AArch64.HaveS1TG()
// ==================
// Determine whether the given translation granule is supported for stage 1

boolean AArch64.HaveS1TG(TGx tgx)
    case tgx of
        when TGx_4KB  return IsFeatureImplemented(FEAT_TGran4K);
        when TGx_16KB return IsFeatureImplemented(FEAT_TGran16K);
        when TGx_64KB return IsFeatureImplemented(FEAT_TGran64K);
```

G1.1.4.55 AArch64.MaxTxSZ

```
// AArch64.MaxTxSZ()
// =================
// Retrieve the maximum value of TxSZ indicating minimum input address size for both
// stages of translation

integer AArch64.MaxTxSZ(TGx tgx)
    if IsFeatureImplemented(FEAT_TTST) then
        case tgx of
            when TGx_4KB   return 48;
            when TGx_16KB  return 48;
            when TGx_64KB  return 47;

    return 39;
```

G1.1.4.56 AArch64.PAMax

```
// AArch64.PAMax()
// ===============
// Returns the IMPLEMENTATION DEFINED maximum number of bits capable of representing
// physical address for this PE

AddressSize AArch64.PAMax()
    return integer IMPLEMENTATION_DEFINED "Maximum Physical Address Size";
```

G1.1.4.57 AArch64.S1BREnabled

```
// AArch64.S1BREnabled()
// =====================
// Determine the number of MPUs supported in PMSA stage 1 for given regime

boolean AArch64.S1BREnabled(Regime regime)
    case regime of
        when Regime_EL2
            return SCTLR_EL2.BR == '1';
        when Regime_EL10
            return HCR_EL2.<DC,TGE> == '00' && VTCR_EL2.MSA == '0' && SCTLR_EL1.BR == '1';
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.1.4.58 AArch64.S1DCacheEnabled

```
// AArch64.S1DCacheEnabled()
// =========================
// Determine cacheability of stage 1 data accesses

boolean AArch64.S1DCacheEnabled(Regime regime)
    case regime of
        when Regime_EL2  return SCTLR_EL2.C == '1';
        when Regime_EL10 return SCTLR_EL1.C == '1';
```

G1.1.4.59 AArch64.S1DecodeTG0

```
// AArch64.S1DecodeTG0()
// =====================
// Decode stage 1 granule size configuration bits TG0

TGx AArch64.S1DecodeTG0(bits(2) tg0_in)
    bits(2) tg0 = tg0_in;
    TGx tgx;

    if tg0 == '11' then
        tg0 = bits(2) IMPLEMENTATION_DEFINED "TG0 encoded granule size";

    case tg0 of
        when '00'   tgx = TGx_4KB;
        when '01'   tgx = TGx_64KB;
        when '10'   tgx = TGx_16KB;

    if !AArch64.HaveS1TG(tgx) then
        case bits(2) IMPLEMENTATION_DEFINED "TG0 encoded granule size" of
            when '00'   tgx = TGx_4KB;
            when '01'   tgx = TGx_64KB;
            when '10'   tgx = TGx_16KB;

    return tgx;
```

G1.1.4.60 AArch64.S1DecodeTG1

```
// AArch64.S1DecodeTG1()
// =====================
// Decode stage 1 granule size configuration bits TG1

TGx AArch64.S1DecodeTG1(bits(2) tg1_in)
    bits(2) tg1 = tg1_in;
    TGx tgx;

    if tg1 == '00' then
        tg1 = bits(2) IMPLEMENTATION_DEFINED "TG1 encoded granule size";

    case tg1 of
        when '10'   tgx = TGx_4KB;
        when '11'   tgx = TGx_64KB;
        when '01'   tgx = TGx_16KB;

    if !AArch64.HaveS1TG(tgx) then
        case bits(2) IMPLEMENTATION_DEFINED "TG1 encoded granule size" of
            when '10'   tgx = TGx_4KB;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            when '11'   tgx = TGx_64KB;
            when '01'   tgx = TGx_16KB;

    return tgx;
```

G1.1.4.61 AArch64.S1EPD

```
// AArch64.S1EPD()
// ===============
// Determine whether stage 1 translation table walk is allowed for the VA range

bit AArch64.S1EPD(Regime regime, bits(64) va)
    assert HasUnprivileged(regime);
    varange = AArch64.GetVARange(va);

    case regime of
        when Regime_EL10 return if varange == VARange_LOWER then TCR_EL1.EPD0 else TCR_EL1.EPD1;
```

G1.1.4.62 AArch64.S1Enabled

```
// AArch64.S1Enabled()
// ===================
// Determine if stage 1 is enabled for the access type for this translation regime

boolean AArch64.S1Enabled(Regime regime, AccessType acctype)
    case regime of
        when Regime_EL2  return SCTLR_EL2.M == '1';
        when Regime_EL10 return (!EL2Enabled() || HCR_EL2.<DC,TGE> == '00') && SCTLR_EL1.M == '1';
```

G1.1.4.63 AArch64.S1ICacheEnabled

```
// AArch64.S1ICacheEnabled()
// =========================
// Determine cacheability of stage 1 instruction fetches

boolean AArch64.S1ICacheEnabled(Regime regime)
    case regime of
        when Regime_EL2  return SCTLR_EL2.I == '1';
        when Regime_EL10 return SCTLR_EL1.I == '1';
```

G1.1.4.64 AArch64.S1MinTxSZ

```
// AArch64.S1MinTxSZ()
// ===================
// Retrieve the minimum value of TxSZ indicating maximum input address size for stage 1

integer AArch64.S1MinTxSZ(Regime regime, S1TTWParams walkparams)
    if IsFeatureImplemented(FEAT_LVA) then
        if walkparams.tgx == TGx_64KB then
            return 12;
        else
            return 16;

    return 16;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.1.4.65 AArch64.S1TTBR

```
// AArch64.S1TTBR()
// ================
// Identify stage 1 table base register for the acting translation regime

bits(64) AArch64.S1TTBR(Regime regime, bits(64) va)
    varange = AArch64.GetVARange(va);

    case regime of
        when Regime_EL10 return if varange == VARange_LOWER then TTBR0_EL1 else TTBR1_EL1;
```

G1.1.4.66 AArch64.S1TTWParamsEL10

```
// AArch64.S1TTWParamsEL10()
// =========================
// Gather stage 1 translation table walk parameters for EL1&0 regime
// (with EL2 enabled or disabled)

S1TTWParams AArch64.S1TTWParamsEL10(bits(2) el, VARange varange)
    S1TTWParams walkparams;

    if varange == VARange_LOWER then
        walkparams.tgx   = AArch64.S1DecodeTG0(TCR_EL1.TG0);
        walkparams.txsz  = TCR_EL1.T0SZ;
        walkparams.irgn  = TCR_EL1.IRGN0;
        walkparams.orgn  = TCR_EL1.ORGN0;
        walkparams.sh    = TCR_EL1.SH0;
        walkparams.tbi   = TCR_EL1.TBI0;

        walkparams.tbid  = if IsFeatureImplemented(FEAT_PAuth) then TCR_EL1.TBID0 else '0';
        walkparams.e0pd  = if IsFeatureImplemented(FEAT_E0PD)  then TCR_EL1.E0PD0 else '0';
        walkparams.hpd   = if IsFeatureImplemented(FEAT_HPDS)  then TCR_EL1.HPD0  else '0';
    else
        walkparams.tgx   = AArch64.S1DecodeTG1(TCR_EL1.TG1);
        walkparams.txsz  = TCR_EL1.T1SZ;
        walkparams.irgn  = TCR_EL1.IRGN1;
        walkparams.orgn  = TCR_EL1.ORGN1;
        walkparams.sh    = TCR_EL1.SH1;
        walkparams.tbi   = TCR_EL1.TBI1;

        walkparams.tbid  = if IsFeatureImplemented(FEAT_PAuth) then TCR_EL1.TBID1 else '0';
        walkparams.e0pd  = if IsFeatureImplemented(FEAT_E0PD)  then TCR_EL1.E0PD1 else '0';
        walkparams.hpd   = if IsFeatureImplemented(FEAT_HPDS)  then TCR_EL1.HPD1  else '0';

    walkparams.mair = MAIR_EL1;
    walkparams.wxn  = SCTLR_EL1.WXN;
    walkparams.ps   = TCR_EL1.IPS;
    walkparams.ee   = SCTLR_EL1.EE;

    if EL2Enabled() then
        walkparams.dc  = HCR_EL2.DC;

    walkparams.ha   = if IsFeatureImplemented(FEAT_HAFDBS) then TCR_EL1.HA else '0';
    walkparams.hd   = if walkparams.ha == '1' then TCR_EL1.HD else '0';

    return walkparams;
```

G1.1.4.67 S2DCacheEnabled

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// S2DCacheEnabled()
// =================
// Returns TRUE if Stage 2 Data access cacheability is enabled

boolean S2DCacheEnabled()
    return HCR_EL2.CD == '0';
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2 Shared pseudocode

This section holds the pseudocode that is common to execution in AArch64 state and in AArch32 state. Functions listed in this section are identified only by a `FunctionName`, without an `AArch64.` or `AArch32.` prefix. This section is organized by functional groups, with the functional groups being indicated by hierarchical path names, for example `shared/debug/DebugTarget`.

The top-level sections of the shared pseudocode hierarchy are:

- shared/debug.
- shared/exceptions.
- shared/functions.
- shared/trace.
- shared/translation.

G1.2.1 shared/debug

This section includes the following pseudocode functions:

G1.2.1.1 ClearStickyErrors

```
// ClearStickyErrors()
// ===================

ClearStickyErrors()
    EDSCR.TXU = '0';            // Clear TX underrun flag
    EDSCR.RXO = '0';            // Clear RX overrun flag

    if Halted() then            // in Debug state
        EDSCR.ITO = '0';        // Clear ITR overrun flag

    // If halted and the ITR is not empty then it is UNPREDICTABLE whether the EDSCR.ERR is cleared.
    // The UNPREDICTABLE behavior also affects the instructions in flight, but this is not described
    // in the pseudocode.
    if (Halted() && EDSCR.ITE == '0' &&
          ConstrainUnpredictableBool(Unpredictable_CLEARERRITEZERO)) then
        return;
    EDSCR.ERR = '0';            // Clear cumulative error flag

    return;
```

G1.2.1.2 DebugTarget

```
// DebugTarget()
// =============
// Returns the debug exception target Exception level

bits(2) DebugTarget()
    ss = CurrentSecurityState();
    return DebugTargetFrom(ss);
```

G1.2.1.3 DebugTargetFrom

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// DebugTargetFrom()
// =================

bits(2) DebugTargetFrom(SecurityState from_state)
    boolean route_to_el2;
    if from_state != SS_Secure || IsFeatureImplemented(FEAT_SEL2) then
        route_to_el2 = (MDCR_EL2.TDE == '1' || HCR_EL2.TGE == '1');
    else
        route_to_el2 = FALSE;

    bits(2) target;
    if route_to_el2 then
        target = EL2;
    elsif HaveEL(EL3) && !HaveAArch64() && from_state == SS_Secure then
        target = EL3;
    else
        target = EL1;

    return target;
```

G1.2.1.4 DoubleLockStatus

```
// DoubleLockStatus()
// ==================
// Returns the state of the OS Double Lock.
//    FALSE if OSDLR_EL1.DLK == 0 or DBGPRCR_EL1.CORENPDRQ == 1 or the PE is in Debug state.
//    TRUE if OSDLR_EL1.DLK == 1 and DBGPRCR_EL1.CORENPDRQ == 0 and the PE is in Non-debug state.

boolean DoubleLockStatus()
    if !IsFeatureImplemented(FEAT_DoubleLock) then
        return FALSE;
    return OSDLR_EL1.DLK == '1' && DBGPRCR_EL1.CORENPDRQ == '0' && !Halted();
```

G1.2.1.5 OSLockStatus

```
// OSLockStatus()
// ==============
// Returns the state of the OS Lock.

boolean OSLockStatus()
    return OSLSR_EL1.OSLK == '1';
```

G1.2.1.6 Component

```
// Component
// =========
// Component Types.

enumeration Component {
        Component_RAS,
        Component_GIC,
        Component_PMU,
        Component_Debug,
        Component_CTI
};
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.1.7 GetAccessComponent

```
// GetAccessComponent()
// ====================
// Returns the accessed component.

Component GetAccessComponent();
```

G1.2.1.8 SoftwareLockStatus

```
// SoftwareLockStatus()
// ====================
// Returns the state of the Software Lock.

boolean SoftwareLockStatus()
    constant Component component = GetAccessComponent();
    if !HaveSoftwareLock(component) then
        return FALSE;
    case component of
        when Component_Debug
            return EDLSR.SLK == '1';
        when Component_PMU
            return PMLSR.SLK == '1';
        when Component_CTI
            return CTILSR.SLK == '1';
        otherwise
            return FALSE;
```

G1.2.1.9 AllowExternalDebugAccess

```
// AllowExternalDebugAccess()
// ==========================
// Returns TRUE if an external debug interface access to the External DBGBVR<n>_EL1,
// DBGBCR<n>_EL1, DBGWVR<n>_EL1, DBGWCR<n>_EL1 registers, and, from Armv8.2, the
// OSLAR_EL1 register is allowed for the access. Returns FALSE otherwise.

boolean AllowExternalDebugAccess(AddressDescriptor addrdesc)
    // The access may also be subject to OS Lock, power-down, etc.
    if IsFeatureImplemented(FEAT_Debugv8p4) then
        if addrdesc.paddress.paspace == PAS_Secure then return TRUE;
    else
        if !ExternalInvasiveDebugEnabled() then return FALSE;
        if ExternalSecureInvasiveDebugEnabled() then return TRUE;

    Unreachable();
```

G1.2.1.10 AllowExternalPMUAccess

```
// AllowExternalPMUAccess()
// ========================
// Returns TRUE if an external debug interface access to the PMU registers is
// allowed for the given Security state, FALSE otherwise.

boolean AllowExternalPMUAccess(AddressDescriptor addrdesc)
    // The access may also be subject to OS Lock, power-down, etc.
    if IsFeatureImplemented(FEAT_Debugv8p4) then
        if addrdesc.paddress.paspace == PAS_Secure then return TRUE;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        if !ExternalInvasiveDebugEnabled() then return FALSE;
        if ExternalSecureInvasiveDebugEnabled() then return TRUE;

    Unreachable();
```

G1.2.1.11 Debug

```
// Debug authentication signals
// ============================

Signal DBGEN;
Signal NIDEN;
Signal SPIDEN;
Signal SPNIDEN;
```

G1.2.1.12 ExternalInvasiveDebugEnabled

```
// ExternalInvasiveDebugEnabled()
// ==============================
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the DBGEN signal.

boolean ExternalInvasiveDebugEnabled()
    return DBGEN == HIGH;
```

G1.2.1.13 ExternalNoninvasiveDebugAllowed

```
// ExternalNoninvasiveDebugAllowed()
// =================================
// Returns TRUE if Trace and PC Sample-based Profiling are allowed

boolean ExternalNoninvasiveDebugAllowed()
    return ExternalNoninvasiveDebugAllowed(PSTATE.EL);

// ExternalNoninvasiveDebugAllowed()
// =================================

boolean ExternalNoninvasiveDebugAllowed(bits(2) el)
    if !ExternalNoninvasiveDebugEnabled() then return FALSE;
    ss = SecurityStateAtEL(el);

    case ss of
        when SS_NonSecure return TRUE;
        when SS_Secure    return ExternalSecureNoninvasiveDebugEnabled();
```

G1.2.1.14 ExternalNoninvasiveDebugEnabled

```
// ExternalNoninvasiveDebugEnabled()
// =================================
// This function returns TRUE if the FEAT_Debugv8p4 is implemented.
// Otherwise, this function is IMPLEMENTATION DEFINED, and, in the
// recommended interface, ExternalNoninvasiveDebugEnabled returns
// the state of the (DBGEN OR NIDEN) signal.
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
boolean ExternalNoninvasiveDebugEnabled()
    return (IsFeatureImplemented(FEAT_Debugv8p4) || ExternalInvasiveDebugEnabled() ||
            NIDEN == HIGH);
```

G1.2.1.15 ExternalSecureInvasiveDebugEnabled

```
// ExternalSecureInvasiveDebugEnabled()
// ====================================
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the (DBGEN AND SPIDEN) signal.
// CoreSight allows asserting SPIDEN without also asserting DBGEN, but this is not recommended.

boolean ExternalSecureInvasiveDebugEnabled()
    if !HaveSecureState() then return FALSE;
    return ExternalInvasiveDebugEnabled() && SPIDEN == HIGH;
```

G1.2.1.16 ExternalSecureNoninvasiveDebugEnabled

```
// ExternalSecureNoninvasiveDebugEnabled()
// =======================================
// This function returns the value of ExternalSecureInvasiveDebugEnabled() when FEAT_Debugv8p4
// is implemented. Otherwise, the definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the (DBGEN OR NIDEN) AND
// (SPIDEN OR SPNIDEN) signal.

boolean ExternalSecureNoninvasiveDebugEnabled()
    if !HaveSecureState() then return FALSE;
    if !IsFeatureImplemented(FEAT_Debugv8p4) then
        return (ExternalNoninvasiveDebugEnabled() &&
                (SPIDEN == HIGH || SPNIDEN == HIGH));
    else
        return ExternalSecureInvasiveDebugEnabled();
```

G1.2.1.17 IsAccessNonSecure

```
// IsAccessNonSecure()
// ===================
// Returns TRUE when an access is Non-Secure

boolean IsAccessNonSecure(AddressDescriptor addrdesc)
    return addrdesc.paddress.paspace == PAS_NonSecure;
```

G1.2.1.18 IsAccessSecure

```
// IsAccessSecure()
// ================
// Returns TRUE when an access is Secure

boolean IsAccessSecure(AddressDescriptor addrdesc)
    return addrdesc.paddress.paspace == PAS_Secure;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.1.19 IsCorePowered

```
// IsCorePowered()
// ===============
// Returns TRUE if the Core power domain is powered on, FALSE otherwise.

boolean IsCorePowered();
```

G1.2.1.20 BreakpointInfo

```
// BreakpointInfo
// ==============
// Breakpoint related fields.

type BreakpointInfo is (
    boolean match,         // breakpoint match
    boolean mismatch       // breakpoint mismatch
)
```

G1.2.1.21 CheckValidStateMatch

```
// CheckValidStateMatch()
// ======================
// Checks for an invalid state match that will generate Constrained
// Unpredictable behavior, otherwise returns Constraint_NONE.

(Constraint, bits(2), bit, bits(2)) CheckValidStateMatch(bits(2) ssc_in, bit hmc_in, bits(2) pxc_in,
                                                         boolean isbreakpnt)
    boolean reserved = FALSE;
    bits(2) ssc = ssc_in;
    bit hmc     = hmc_in;
    bits(2) pxc = pxc_in;

    // Values that are not allocated in any architecture version
    if (hmc:ssc:pxc) IN {'01110','100x0','10110','11x10'} then
        reserved = TRUE;

    // Match 'Usr/Sys/Svc' valid only for AArch32 breakpoints
    if hmc:pxc == '000' && ssc != '11' then
        reserved = TRUE;

    // Both EL3 and EL2 are not implemented
    if !HaveEL(EL3) && !HaveEL(EL2) && (hmc != '0' || ssc != '00') then
        reserved = TRUE;

    // EL3 is not implemented
    if !HaveEL(EL3) && ssc IN {'01','10'} && hmc:ssc:pxc != '10100' then
        reserved = TRUE;

    // EL3 using AArch64 only
    if (!HaveEL(EL3) || !HaveAArch64()) && hmc:ssc:pxc == '11000' then
        reserved = TRUE;

    // EL2 is not implemented
    if !HaveEL(EL2) && hmc:ssc:pxc == '11100' then
        reserved = TRUE;

    // Secure EL2 is not implemented
    if !IsFeatureImplemented(FEAT_SEL2) && (hmc:ssc:pxc)  IN {'01100','10100','x11x1'} then
        reserved = TRUE;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if reserved then
        // If parameters are set to a reserved type, behaves as either disabled or a defined type
        Constraint c;
        bits(5) unpred_state_bits;
        (c, unpred_state_bits) = ConstrainUnpredictableBits(Unpredictable_RESBPWPCTRL, 5);
        hmc = unpred_state_bits<4>;
        ssc = unpred_state_bits<3:2>;
        pxc = unpred_state_bits<1:0>;
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then
            return (c, bits(2) UNKNOWN, bit UNKNOWN, bits(2) UNKNOWN);
        // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

    return (Constraint_NONE, ssc, hmc, pxc);
```

G1.2.1.22 ContextAwareBreakpointRange

```
// ContextAwareBreakpointRange()
// =============================
// Returns two numbers indicating the index of the first and last context-aware breakpoint.

(integer, integer) ContextAwareBreakpointRange()
    constant integer b = NumBreakpointsImplemented();
    constant integer c = NumContextAwareBreakpointsImplemented();

    return (b - c, b - 1);
```

G1.2.1.23 IsContextAwareBreakpoint

```
// IsContextAwareBreakpoint()
// ==========================
// Returns TRUE if DBGBCR_EL1[n] is a context-aware breakpoint.

boolean IsContextAwareBreakpoint(integer n)
    (lower, upper) = ContextAwareBreakpointRange();
    return n >= lower && n <= upper;
```

G1.2.1.24 NumBreakpointsImplemented

```
// NumBreakpointsImplemented()
// ===========================
// Returns the number of breakpoints implemented.

integer NumBreakpointsImplemented()
    return integer IMPLEMENTATION_DEFINED "Number of breakpoints";
```

G1.2.1.25 NumContextAwareBreakpointsImplemented

```
// NumContextAwareBreakpointsImplemented()
// =======================================
// Returns the number of context-aware breakpoints implemented.

integer NumContextAwareBreakpointsImplemented()
    return integer IMPLEMENTATION_DEFINED "Number of context-aware breakpoints";
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.1.26 NumWatchpointsImplemented

```
// NumWatchpointsImplemented()
// ===========================
// Returns the number of watchpoints implemented.

integer NumWatchpointsImplemented()
    return integer IMPLEMENTATION_DEFINED "Number of watchpoints";
```

G1.2.1.27 CTI_ProcessEvent

```
// CTI_ProcessEvent()
// ==================
// Process a discrete event on a Cross Trigger output event trigger.

CTI_ProcessEvent(CrossTriggerOut id);
```

G1.2.1.28 CTI_SetEventLevel

```
// CTI_SetEventLevel()
// ===================
// Set a Cross Trigger multi-cycle input event trigger to the specified level.

CTI_SetEventLevel(CrossTriggerIn id, Signal level);
```

G1.2.1.29 CTI_SignalEvent

```
// CTI_SignalEvent()
// =================
// Signal a discrete event on a Cross Trigger input event trigger.

CTI_SignalEvent(CrossTriggerIn id);
```

G1.2.1.30 CrossTrigger

```
// CrossTrigger
// ============

enumeration CrossTriggerOut {CrossTriggerOut_DebugRequest, CrossTriggerOut_RestartRequest,
                             CrossTriggerOut_IRQ,          CrossTriggerOut_RSVD3,
                             CrossTriggerOut_TraceExtIn0,  CrossTriggerOut_TraceExtIn1,
                             CrossTriggerOut_TraceExtIn2,  CrossTriggerOut_TraceExtIn3};

enumeration CrossTriggerIn  {CrossTriggerIn_CrossHalt,     CrossTriggerIn_PMUOverflow,
                             CrossTriggerIn_SPESample,     CrossTriggerIn_RSVD3,
                             CrossTriggerIn_TraceExtOut0,  CrossTriggerIn_TraceExtOut1,
                             CrossTriggerIn_TraceExtOut2,  CrossTriggerIn_TraceExtOut3,
                             CrossTriggerIn_TRBEStop,      CrossTriggerIn_TRBEMgmt,
                             CrossTriggerIn_TRBEWrap};
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.1.31 CheckForDCCInterrupts

```
// CheckForDCCInterrupts()
// =======================

CheckForDCCInterrupts()
    commrx = (EDSCR.RXfull == '1');
    commtx = (EDSCR.TXfull == '0');

    // COMMRX and COMMTX support is optional and not recommended for new designs.
    // SetInterruptRequestLevel(InterruptID_COMMRX, if commrx then HIGH else LOW);
    // SetInterruptRequestLevel(InterruptID_COMMTX, if commtx then HIGH else LOW);

    // The value to be driven onto the common COMMIRQ signal.
    boolean commirq;
    commirq = ((commrx && MDCCINT_EL1.RX == '1') ||
               (commtx && MDCCINT_EL1.TX == '1'));
    SetInterruptRequestLevel(InterruptID_COMMIRQ, if commirq then HIGH else LOW);

    return;
```

G1.2.1.32 DTR

```
// DTR
// ===

bits(32) DTRRX;
bits(32) DTRTX;
```

G1.2.1.33 Read_DBGDTRRX_EL0

```
// Read_DBGDTRRX_EL0()
// ===================
// Called on reads of debug register 0x080.

bits(32) Read_DBGDTRRX_EL0(boolean memory_mapped)
    return DTRRX;
```

G1.2.1.34 Read_DBGDTRTX_EL0

```
// Read_DBGDTRTX_EL0()
// ===================
// Called on reads of debug register 0x08C.

bits(32) Read_DBGDTRTX_EL0(boolean memory_mapped)
    underrun = EDSCR.TXfull == '0' || (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0');
    value = if underrun then bits(32) UNKNOWN else DTRTX;

    if EDSCR.ERR == '1' then return value;              // Error flag set: no side-effects

    if underrun then
        EDSCR.TXU = '1';  EDSCR.ERR = '1';              // Underrun condition: block side-effects
        return value;                                   // Return UNKNOWN

    EDSCR.TXfull = '0';
    if Halted() && EDSCR.MA == '1' then
        EDSCR.ITE = '0';                                // See comments in Write_EDITR()
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        ExecuteA64(0xB8404401<31:0>);               // A64 "LDR W1,[X0],#4"
        // If the load aborts, the Data Abort exception is taken and EDSCR.ERR is set to 1
        if EDSCR.ERR == '1' then
            EDSCR.TXfull = bit UNKNOWN;
            DBGDTRTX_EL0 = bits(64) UNKNOWN;
        else
            ExecuteA64(0xD5130501<31:0>);           // A64 "MSR DBGDTRTX_EL0,X1"
            // "MSR DBGDTRTX_EL0,X1" calls Write_DBGDTR_EL0() which sets TXfull.
            assert EDSCR.TXfull == '1';
        X[1, 64] = bits(64) UNKNOWN;
        EDSCR.ITE = '1';                            // See comments in Write_EDITR()

    return value;
```

G1.2.1.35 Read_DBGDTR_EL0

```
// Read_DBGDTR_EL0()
// =================
// System register reads of DBGDTR_EL0, DBGDTRRX_EL0 (AArch64) and DBGDTRRXint (AArch32)

bits(N) Read_DBGDTR_EL0(integer N)
    // For MRS <Rt>,DBGDTRTX_EL0  N=32, X[t]=Zeros(32):result
    // For MRS <Xt>,DBGDTR_EL0    N=64, X[t]=result
    assert N IN {32,64};
    bits(N) result;
    if EDSCR.RXfull == '0' then
        result = bits(N) UNKNOWN;
    else
        // On a 64-bit read, implement a half-duplex channel
        // NOTE: the word order is reversed on reads with regards to writes
        if N == 64 then result<63:32> = DTRTX;
        result<31:0> = DTRRX;
    EDSCR.RXfull = '0';
    return result;
```

G1.2.1.36 Write_DBGDTRRX_EL0

```
// Write_DBGDTRRX_EL0()
// ====================
// Called on writes to debug register 0x080.

Write_DBGDTRRX_EL0(boolean memory_mapped, bits(32) value)
    if EDSCR.ERR == '1' then return;                    // Error flag set: ignore write

    if EDSCR.RXfull == '1' || (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0') then
        EDSCR.RXO = '1';  EDSCR.ERR = '1';              // Overrun condition: ignore write
        return;

    EDSCR.RXfull = '1';
    DTRRX = value;

    if Halted() && EDSCR.MA == '1' then
        EDSCR.ITE = '0';                                // See comments in Write_EDITR()
        ExecuteA64(0xD5330501<31:0>);                   // A64 "MRS X1,DBGDTRRX_EL0"
        ExecuteA64(0xB8004401<31:0>);                   // A64 "STR W1,[X0],#4"
        X[1, 64] = bits(64) UNKNOWN;

        // If the store aborts, the Data Abort exception is taken and EDSCR.ERR is set to 1
        if EDSCR.ERR == '1' then
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            EDSCR.RXfull = bit UNKNOWN;
            DBGDTRRX_EL0 = bits(64) UNKNOWN;
        else
            // "MRS X1,DBGDTRRX_EL0" calls Read_DBGDTR_EL0() which clears RXfull.
            assert EDSCR.RXfull == '0';

        EDSCR.ITE = '1';                                // See comments in Write_EDITR()
    return;
```

G1.2.1.37 Write_DBGDTRTX_EL0

```
// Write_DBGDTRTX_EL0()
// ====================
// Called on writes to debug register 0x08C.

Write_DBGDTRTX_EL0(boolean memory_mapped, bits(32) value)
    DTRTX = value;
    return;
```

G1.2.1.38 Write_DBGDTR_EL0

```
// Write_DBGDTR_EL0()
// ==================
// System register writes to DBGDTR_EL0, DBGDTRTX_EL0 (AArch64) and DBGDTRTXint (AArch32)

Write_DBGDTR_EL0(bits(N) value_in)
    bits(N) value = value_in;
    // For MSR DBGDTRTX_EL0,<Rt>  N=32, value=X[t]<31:0>, X[t]<63:32> is ignored
    // For MSR DBGDTR_EL0,<Xt>    N=64, value=X[t]<63:0>
    assert N IN {32,64};
    if EDSCR.TXfull == '1' then
        value = bits(N) UNKNOWN;
    // On a 64-bit write, implement a half-duplex channel
    if N == 64 then DTRRX = value<63:32>;
    DTRTX = value<31:0>;        // 32-bit or 64-bit write
    EDSCR.TXfull = '1';
    return;
```

G1.2.1.39 Write_EDITR

```
// Write_EDITR()
// =============
// Called on writes to debug register 0x084.

Write_EDITR(boolean memory_mapped, bits(32) value)
    if EDSCR.ERR == '1' then return;                        // Error flag set: ignore write

    if !Halted() then return;                               // Non-debug state: ignore write

    if EDSCR.ITE == '0' || EDSCR.MA == '1' then
        EDSCR.ITO = '1';  EDSCR.ERR = '1';                  // Overrun condition: block write
        return;

    // ITE indicates whether the PE is ready to accept another instruction; the PE
    // may support multiple outstanding instructions. Unlike the "InstrCompl" flag in [v7A] there
    // is no indication that the pipeline is empty (all instructions have completed). In this
    // pseudocode, the assumption is that only one instruction can be executed at a time,
    // meaning ITE acts like "InstrCompl".
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    EDSCR.ITE = '0';

    ExecuteA64(value);

    EDSCR.ITE = '1';

    return;
```

G1.2.1.40 DCPSInstruction

```
// DCPSInstruction()
// =================
// Operation of the DCPS instruction in Debug state

DCPSInstruction(bits(2) target_el)

    SynchronizeContext();

    bits(2) handle_el;
    case target_el of
        when EL1
            if PSTATE.EL == EL2 || (PSTATE.EL == EL3 && !UsingAArch32()) then
                handle_el = PSTATE.EL;
            elsif EL2Enabled() && HCR_EL2.TGE == '1' then
                UNDEFINED;
            else
                handle_el = EL1;
        when EL2
            if !HaveEL(EL2) then
                UNDEFINED;
            elsif PSTATE.EL == EL3 && !UsingAArch32() then
                handle_el = EL3;
            elsif !IsSecureEL2Enabled() && CurrentSecurityState() == SS_Secure then
                UNDEFINED;
            else
                handle_el = EL2;
        when EL3
            if EDSCR.SDD == '1' || !HaveEL(EL3) then
                UNDEFINED;
            else
                handle_el = EL3;
        otherwise
            Unreachable();

    from_secure = CurrentSecurityState() == SS_Secure;
    PSTATE.nRW = '0';  PSTATE.SP = '1';  PSTATE.EL = handle_el;
    if (IsFeatureImplemented(FEAT_PAN) && handle_el == EL1 && SCTLR_EL1.SPAN == '0') then
        PSTATE.PAN = '1';
    ELR_ELx[] = bits(64) UNKNOWN;  SPSR_ELx[] = bits(64) UNKNOWN;  ESR_ELx[] = bits(64) UNKNOWN;
    DLR_EL0 = bits(64) UNKNOWN;  DSPSR_EL0 = bits(64) UNKNOWN;
    if IsFeatureImplemented(FEAT_UAO) then PSTATE.UAO = '0';
    UpdateEDSCRFields();                          // Update EDSCR PE state flags
    sync_errors = IsFeatureImplemented(FEAT_IESB) && SCTLR_ELx[].IESB == '1';
    // The Effective value of SCTLR[].IESB might be zero in Debug state.
    if !ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
        sync_errors = FALSE;
    if sync_errors then
        SynchronizeErrors();
    return;
```

G1.2.1.41 DRPSInstruction

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// DRPSInstruction()
// =================
// Operation of the A64 DRPS and T32 ERET instructions in Debug state

DRPSInstruction()

    sync_errors = IsFeatureImplemented(FEAT_IESB) && SCTLR_ELx[].IESB == '1';
    // The Effective value of SCTLR[].IESB might be zero in Debug state.
    if !ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
        sync_errors = FALSE;
    if sync_errors then
        SynchronizeErrors();

    SynchronizeContext();

    DebugRestorePSR();

    return;
```

G1.2.1.42 DebugHalt

```
// DebugHalt
// =========
// Reason codes for entry to Debug state

constant bits(6) DebugHalt_Breakpoint      = '000111';
constant bits(6) DebugHalt_EDBGRQ          = '010011';
constant bits(6) DebugHalt_Step_Normal     = '011011';
constant bits(6) DebugHalt_Step_Exclusive  = '011111';
constant bits(6) DebugHalt_OSUnlockCatch   = '100011';
constant bits(6) DebugHalt_ResetCatch      = '100111';
constant bits(6) DebugHalt_Watchpoint      = '101011';
constant bits(6) DebugHalt_HaltInstruction = '101111';
constant bits(6) DebugHalt_SoftwareAccess  = '110011';
constant bits(6) DebugHalt_ExceptionCatch  = '110111';
constant bits(6) DebugHalt_Step_NoSyndrome = '111011';
```

G1.2.1.43 DebugRestorePSR

```
// DebugRestorePSR()
// =================

DebugRestorePSR()
    // PSTATE.{N,Z,C,V,Q,GE,SS,D,A,I,F} are not observable and ignored in Debug state, so
    // behave as if UNKNOWN.
    constant bits(64) spsr = SPSR_ELx[];
    SetPSTATEFromPSR(spsr);
    PSTATE.<N,Z,C,V,SS,D,A,I,F> = bits(9) UNKNOWN;
    DLR_EL0 = bits(64) UNKNOWN;  DSPSR_EL0 = bits(64) UNKNOWN;
    UpdateEDSCRFields();                                // Update EDSCR PE state flags
```

G1.2.1.44 DisableITRAndResumeInstructionPrefetch

```
// DisableITRAndResumeInstructionPrefetch()
// ========================================
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
DisableITRAndResumeInstructionPrefetch();
```

G1.2.1.45 ExecuteA64

```
// ExecuteA64()
// ============
// Execute an A64 instruction in Debug state.

ExecuteA64(bits(32) instr);
```

G1.2.1.46 ExecuteT32

```
// ExecuteT32()
// ============
// Execute a T32 instruction in Debug state.

ExecuteT32(bits(16) hw1, bits(16) hw2);
```

G1.2.1.47 ExitDebugState

```
// ExitDebugState()
// ================

ExitDebugState()
    assert Halted();
    SynchronizeContext();

    // Although EDSCR.STATUS signals that the PE is restarting, debuggers must use EDPRSR.SDR to
    // detect that the PE has restarted.
    EDSCR.STATUS = '000001';                            // Signal restarting
    EDESR<2:0> = '000';                                 // Clear any pending Halting debug events

    bits(64) new_pc;
    bits(64) spsr;

    new_pc = DLR_EL0;
    spsr = DSPSR_EL0;

    constant boolean illegal_psr_state = IllegalExceptionReturn(spsr);
    // If this is an illegal return, SetPSTATEFromPSR() will set PSTATE.IL.
    SetPSTATEFromPSR(spsr);                             // Can update privileged bits, even at EL0

    constant boolean branch_conditional = FALSE;
    if UsingAArch32() then
        if ConstrainUnpredictableBool(Unpredictable_RESTARTALIGNPC) then new_pc<0> = '0';
        // AArch32 branch
        BranchTo(new_pc<31:0>, BranchType_DBGEXIT, branch_conditional);
    else
        // If targeting AArch32 then PC[63:32,1:0] might be set to UNKNOWN.
        if illegal_psr_state && spsr<4> == '1' then
            new_pc<63:32> = bits(32) UNKNOWN;
            new_pc<1:0> = bits(2) UNKNOWN;
        // A type of branch that is never predicted
        BranchTo(new_pc, BranchType_DBGEXIT, branch_conditional);

    (EDSCR.STATUS,EDPRSR.SDR) = ('000010','1');         // Atomically signal restarted
    EDPRSR.HALTED = '0';
    UpdateEDSCRFields();                                // Stop signalling PE state
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    DisableITRAndResumeInstructionPrefetch();

    return;
```

G1.2.1.48 Halt

```
// Halt()
// ======

Halt(bits(6) reason)
    constant boolean is_async = FALSE;
    Halt(reason, is_async);

// Halt()
// ======

Halt(bits(6) reason, boolean is_async)
    CTI_SignalEvent(CrossTriggerIn_CrossHalt);  // Trigger other cores to halt

    constant bits(64) preferred_restart_address = ThisInstrAddr(64);
    bits(64) spsr = GetPSRFromPSTATE(DebugState, 64);

    DLR_EL0 = preferred_restart_address;
    DSPSR_EL0 = spsr;
    EDSCR.ITE = '1';
    EDSCR.ITO = '0';
    EDSCR.MA = '0';

    // In Debug state:
    // * PSTATE.{SS,SSBS,D,A,I,F} are not observable and ignored so behave-as-if UNKNOWN.
    // * PSTATE.{N,Z,C,V,Q,GE,E,M,nRW,EL,SP,DIT} are also not observable, but since these
    //     are not changed on exception entry, this function also leaves them unchanged.
    // * PSTATE.{IT,T} are ignored.
    // * PSTATE.IL is ignored and behave-as-if 0.
    // * PSTATE.{UAO,PAN} are observable and not changed on entry into Debug state.

    if UsingAArch32() then
        PSTATE.<IT,SS,SSBS,A,I,F,T> = bits(14) UNKNOWN;
    else
        PSTATE.<SS,SSBS,D,A,I,F>    = bits(6)  UNKNOWN;

    PSTATE.IL = '0';
    StopInstructionPrefetchAndEnableITR();
    (EDSCR.STATUS,EDPRSR.HALTED) = (reason,'1');
    UpdateEDSCRFields();                        // Update EDSCR PE state flags.
    if !is_async then EndOfInstruction();
    return;
```

G1.2.1.49 HaltOnBreakpointOrWatchpoint

```
// HaltOnBreakpointOrWatchpoint()
// ==============================
// Returns TRUE if the Breakpoint and Watchpoint debug events should be considered for Debug
// state entry, FALSE if they should be considered for a debug exception.

boolean HaltOnBreakpointOrWatchpoint()
    return HaltingAllowed() && EDSCR.HDE == '1' && OSLSR_EL1.OSLK == '0';
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.1.50 Halted

```
// Halted()
// ========

boolean Halted()
    return ! EDSCR.STATUS IN {'000001', '000010'};                    // Halted
```

G1.2.1.51 HaltingAllowed

```
// HaltingAllowed()
// ================
// Returns TRUE if halting is currently allowed, FALSE if halting is prohibited.

boolean HaltingAllowed()
    if Halted() || DoubleLockStatus() then
        return FALSE;
    ss = CurrentSecurityState();
    case ss of
        when SS_NonSecure return ExternalInvasiveDebugEnabled();
        when SS_Secure    return ExternalSecureInvasiveDebugEnabled();
```

G1.2.1.52 Restarting

```
// Restarting()
// ============

boolean Restarting()
    return EDSCR.STATUS == '000001';                                  // Restarting
```

G1.2.1.53 StopInstructionPrefetchAndEnableITR

```
// StopInstructionPrefetchAndEnableITR()
// =====================================

StopInstructionPrefetchAndEnableITR();
```

G1.2.1.54 UpdateDbgAuthStatus

```
// UpdateDbgAuthStatus()
// =====================
// Provides information about the state of the
// IMPLEMENTATION DEFINED authentication interface for debug.

UpdateDbgAuthStatus()
    bits(2) nsid, nsnid;
    bits(2) sid, snid;
    if SecureOnlyImplementation() then
        nsid = '00';
    elsif ExternalInvasiveDebugEnabled() then
        nsid = '11';          // Non-secure Invasive debug implemented and enabled.
    else
        nsid = '10';          // Non-secure Invasive debug implemented and disabled.
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if SecureOnlyImplementation() then
        nsnid = '00';
    elsif ExternalNoninvasiveDebugEnabled() then
        nsnid = '11';          // Non-secure Non-Invasive debug implemented and enabled.
    else
        nsnid = '10';          // Non-secure Non-Invasive debug implemented and disabled.

    if !HaveSecureState() then
        sid = '00';
    elsif ExternalSecureInvasiveDebugEnabled() then
        sid = '11';            // Secure Invasive debug implemented and enabled.
    else
        sid = '10';            // Secure Invasive debug implemented and disabled.

    if !HaveSecureState() then
        snid = '00';
    elsif ExternalSecureNoninvasiveDebugEnabled() then
        snid = '11';           // Secure Non-Invasive debug implemented and enabled.
    else
        snid = '10';           // Secure Non-Invasive debug implemented and disabled.

    DBGAUTHSTATUS_EL1.NSID  = nsid;
    DBGAUTHSTATUS_EL1.NSNID = nsnid;
    DBGAUTHSTATUS_EL1.SID   = sid;
    DBGAUTHSTATUS_EL1.SNID  = snid;
    return;
```

G1.2.1.55 UpdateEDSCRFields

```
// UpdateEDSCRFields()
// ===================
// Update EDSCR PE state fields

UpdateEDSCRFields()
    if !Halted() then
        EDSCR.EL = '00';
        EDSCR.SDD = if ExternalSecureInvasiveDebugEnabled() then '0' else '1';
        EDSCR.NS = bit UNKNOWN;

        EDSCR.RW = '1111';
    else
        EDSCR.EL = PSTATE.EL;
        EDSCR.NS = '0';
        EDSCR.RW = '1111';
    return;
```

G1.2.1.56 CheckEDBGRQ

```
// CheckEDBGRQ()
// =============
// Checks IMPLEMENTATION DEFINED EDBGRQ Input Signal of external debug interface.
// This is an example of an IMPLEMENTATION DEFINED source of External Debug Request debug events.

CheckEDBGRQ()
    if EDBGRQ == HIGH then
        ExternalDebugRequest();
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.1.57 CheckExceptionCatch

```
// CheckExceptionCatch()
// =====================
// Check whether an Exception Catch debug event is set on the current Exception level

CheckExceptionCatch(boolean exception_entry)
    // Called after an exception entry or exit, that is, such that the Security state
    // and PSTATE.EL are correct for the exception target. When FEAT_Debugv8p2
    // is not implemented, this function might also be called at any time.
    ss = SecurityStateAtEL(PSTATE.EL);
    base = if ss == SS_Secure then 0 else 4;
    if HaltingAllowed() then
        boolean halt;
        if IsFeatureImplemented(FEAT_Debugv8p2) then
            exception_exit = !exception_entry;
            increment = 8;
            ctrl = EDECCR<UInt(PSTATE.EL) + base + increment>:EDECCR<UInt(PSTATE.EL) + base>;
            case ctrl of
                when '00'  halt = FALSE;
                when '01'  halt = TRUE;
                when '10'  halt = (exception_exit == TRUE);
                when '11'  halt = (exception_entry == TRUE);
        else
            halt = (EDECCR<UInt(PSTATE.EL) + base> == '1');

        if halt then
            Halt(DebugHalt_ExceptionCatch);
```

G1.2.1.58 CheckExternalDebugRequestEvents

```
// CheckExternalDebugRequestEvents()
// =================================
// Checks for all External Debug Request debug events.

CheckExternalDebugRequestEvents()
    CheckEDBGRQ();
    return;
```

G1.2.1.59 CheckHaltingStep

```
// CheckHaltingStep()
// ==================
// Check whether EDESR.SS has been set by Halting Step

CheckHaltingStep(boolean is_async)
    step_enabled = EDECR.SS == '1' && HaltingAllowed();
    active_pending = step_enabled && EDESR.SS == '1';
    if active_pending then
        if HaltingStep_DidNotStep() then
            Halt(DebugHalt_Step_NoSyndrome, is_async);
        elsif HaltingStep_SteppedEX() then
            Halt(DebugHalt_Step_Exclusive, is_async);
        else
            Halt(DebugHalt_Step_Normal, is_async);
    if step_enabled then ShouldAdvanceHS = TRUE;
    return;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.1.60 CheckOSUnlockCatch

```
// CheckOSUnlockCatch()
// ====================
// Called on unlocking the OS Lock to pend an OS Unlock Catch debug event

CheckOSUnlockCatch()
    if ((IsFeatureImplemented(FEAT_DoPD) && CTIDEVCTL.OSUCE == '1') ||
          (!IsFeatureImplemented(FEAT_DoPD) && EDECR.OSUCE == '1')) then
        if !Halted() then EDESR.OSUC = '1';
```

G1.2.1.61 CheckPendingOSUnlockCatch

```
// CheckPendingOSUnlockCatch()
// ===========================
// Check whether EDESR.OSUC has been set by an OS Unlock Catch debug event

CheckPendingOSUnlockCatch()
    if HaltingAllowed() && EDESR.OSUC == '1' then
        constant boolean is_async = TRUE;
        Halt(DebugHalt_OSUnlockCatch, is_async);
```

G1.2.1.62 CheckPendingResetCatch

```
// CheckPendingResetCatch()
// ========================
// Check whether EDESR.RC has been set by a Reset Catch debug event

CheckPendingResetCatch()
    if HaltingAllowed() && EDESR.RC == '1' then
        constant boolean is_async = TRUE;
        Halt(DebugHalt_ResetCatch, is_async);
```

G1.2.1.63 CheckResetCatch

```
// CheckResetCatch()
// =================
// Called after reset

CheckResetCatch()
    if ((IsFeatureImplemented(FEAT_DoPD) && CTIDEVCTL.RCE == '1') ||
          (!IsFeatureImplemented(FEAT_DoPD) && EDECR.RCE == '1')) then
        EDESR.RC = '1';
        // If halting is allowed then halt immediately
        if HaltingAllowed() then Halt(DebugHalt_ResetCatch);
```

G1.2.1.64 CheckSoftwareAccessToDebugRegisters

```
// CheckSoftwareAccessToDebugRegisters()
// =====================================
// Check for access to Breakpoint and Watchpoint registers.

CheckSoftwareAccessToDebugRegisters()
    os_lock = OSLSR_EL1.OSLK;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if HaltingAllowed() && EDSCR.TDA == '1' && os_lock == '0' then
        Halt(DebugHalt_SoftwareAccess);
```

G1.2.1.65 EDBGRQ

```
Signal EDBGRQ;  // Input Signal in external debug interface
```

G1.2.1.66 ExternalDebugRequest

```
// ExternalDebugRequest()
// ======================

ExternalDebugRequest()
    if HaltingAllowed() then
        constant boolean is_async = TRUE;
        Halt(DebugHalt_EDBGRQ, is_async);
    // Otherwise the CTI continues to assert the debug request until it is taken.
```

G1.2.1.67 HSAdvance

```
// HSAdvance()
// ===========
// Advance the Halting Step State Machine

HSAdvance()
    if !ShouldAdvanceHS then return;
    step_enabled = EDECR.SS == '1' && HaltingAllowed();
    active_not_pending = step_enabled && EDESR.SS == '0';
    if active_not_pending then EDESR.SS = '1';      // set as pending.
    ShouldAdvanceHS = FALSE;
    return;
```

G1.2.1.68 HaltingStep_DidNotStep

```
// HaltingStep_DidNotStep()
// ========================
// Returns TRUE if the previously executed instruction was executed in the inactive state, that is,
// if it was not itself stepped.

boolean HaltingStep_DidNotStep();
```

G1.2.1.69 HaltingStep_SteppedEX

```
// HaltingStep_SteppedEX()
// =======================
// Returns TRUE if the previously executed instruction was a Load-Exclusive class instruction
// executed in the active-not-pending state.

boolean HaltingStep_SteppedEX();
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.1.70 ExternalDebugInterruptsDisabled

```
// ExternalDebugInterruptsDisabled()
// =================================
// Determine whether EDSCR disables interrupts routed to 'target'.

boolean ExternalDebugInterruptsDisabled(bits(2) target)
    boolean int_dis;
    constant SecurityState ss = SecurityStateAtEL(target);
    if IsFeatureImplemented(FEAT_Debugv8p4) then
        if EDSCR.INTdis<0> == '1' then
            case ss of
                when SS_NonSecure int_dis = ExternalInvasiveDebugEnabled();
                when SS_Secure    int_dis = ExternalSecureInvasiveDebugEnabled();
        else
            int_dis = FALSE;
    else
        case target of
            when EL3
                int_dis = (EDSCR.INTdis == '11' && ExternalSecureInvasiveDebugEnabled());
            when EL2
                int_dis = (EDSCR.INTdis == '1x' && ExternalInvasiveDebugEnabled());
            when EL1
                if ss == SS_Secure then
                    int_dis = (EDSCR.INTdis == '1x' && ExternalSecureInvasiveDebugEnabled());
                else
                    int_dis = (EDSCR.INTdis != '00' && ExternalInvasiveDebugEnabled());
    return int_dis;
```

G1.2.1.71 pmu

```
array integer PMUEventAccumulator[0..30];  // Accumulates PMU events for a cycle
```

G1.2.1.72 CYCLE_COUNTER_ID

```
// Constant used in PMU functions to represent actions on the cycle counter.
constant integer CYCLE_COUNTER_ID = 31;
```

G1.2.1.73 CheckForPMUOverflow

```
// CheckForPMUOverflow()
// =====================
// Called before each instruction is executed.
// If a PMU event counter has overflowed, this function might do any of:
//  - Signal a Performance Monitors overflow interrupt request.
//  - Signal a CTI Performance Monitors overflow event.

CheckForPMUOverflow()
    constant boolean include_r1 = TRUE;
    constant boolean include_r2 = TRUE;

    constant boolean pmuirq = CheckPMUOverflowCondition(PMUOverflowCondition_IRQ,
                                                        include_r1, include_r2);

    SetInterruptRequestLevel(InterruptID_PMUIRQ,
                             if pmuirq then HIGH else LOW);
    CTI_SetEventLevel(CrossTriggerIn_PMUOverflow,
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                        if pmuirq then HIGH else LOW);

    return;
```

G1.2.1.74 CheckPMUOverflowCondition

```
// CheckPMUOverflowCondition()
// ===========================
// Checks for PMU overflow under certain parameter conditions described by 'reason'.
// If 'include_r1' is TRUE, then check counters in the range [0..(HPMN-1)] and CCNTR
//     unless excluded by 'reason'.
// If 'include_r2' is TRUE, then check counters in the range [HPMN..(N-1)].

boolean CheckPMUOverflowCondition(PMUOverflowCondition reason,
                                  boolean include_r1, boolean include_r2)

    // 'reason' is decoded into a further set of parameters:
    // If 'check_e' is TRUE, then check the applicable one of PMCR_EL0.E and MDCR_EL2.HPME.
    // If 'check_inten' is TRUE, then check the applicable PMINTENCLR_EL1 bit.
    // If 'exclude_cyc' is TRUE, then CCNTR is NOT checked.
    boolean check_e;
    boolean check_inten;
    boolean exclude_cyc;

    case reason of
        when PMUOverflowCondition_IRQ
            check_e      = TRUE;
            check_inten  = TRUE;
            exclude_cyc  = FALSE;
        otherwise
            Unreachable();

    bits(64) ovsf;

    ovsf = PMOVSSET_EL0;
    ovsf<63:32> = Zeros(32);

    constant integer counters = NUM_PMU_COUNTERS;
    // Remove unimplemented counters - these fields are RES0
    if counters < 31 then
        ovsf<30:counters> = Zeros(31-counters);

    for idx = 0 to counters - 1
        bit global_en;
        case GetPMUCounterRange(idx) of
            when PMUCounterRange_R1
                global_en = PMCR_EL0.E;
                if !include_r1 then
                    ovsf<idx> = '0';
            when PMUCounterRange_R2
                global_en = MDCR_EL2.HPME;
                if !include_r2 then
                    ovsf<idx> = '0';
            otherwise
                Unreachable();
        if check_e then
            ovsf<idx> = ovsf<idx> AND global_en;

    // Cycle counter
    if exclude_cyc || !include_r1 then
        ovsf<CYCLE_COUNTER_ID> = '0';

    if check_e then
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        ovsf<CYCLE_COUNTER_ID> = ovsf<CYCLE_COUNTER_ID> AND PMCR_EL0.E;

    if check_inten then
        constant bits(64) inten = PMINTENCLR_EL1;
        ovsf = ovsf AND inten;

    return !IsZero(ovsf);
```

G1.2.1.75 ClearEventCounters

```
// ClearEventCounters()
// ====================
// Zero all the event counters.
// Called on a write to PMCR_EL0 or PMCR that writes '1' to PMCR_EL0.P or PMCR.P.

ClearEventCounters()
    ZeroPMUCounters(Zeros(33) : Ones(31));
```

G1.2.1.76 CountPMUEvents

```
// CountPMUEvents()
// ================
// Return TRUE if counter "idx" should count its event.
// For the cycle counter, idx == CYCLE_COUNTER_ID (31).

boolean CountPMUEvents(integer idx)
    constant integer counters = NUM_PMU_COUNTERS;
    assert idx == CYCLE_COUNTER_ID || idx < counters;

    boolean debug;
    boolean enabled;
    boolean prohibited;
    boolean filtered;

    // Event counting is disabled in Debug state
    debug = Halted();

    // Software can reserve some counters for EL2
    constant PMUCounterRange counter_range = GetPMUCounterRange(idx);
    ss = CurrentSecurityState();

    // Main enable controls
    bit global_en;
    bit counter_en;
    case counter_range of
        when PMUCounterRange_R1
            global_en = PMCR_EL0.E;
        when PMUCounterRange_R2
            global_en = MDCR_EL2.HPME;
        otherwise
            Unreachable();

    case idx of
        when CYCLE_COUNTER_ID
            counter_en = PMCNTENSET_EL0.C;
        otherwise
            counter_en = PMCNTENSET_EL0<idx>;

    enabled = global_en == '1' && counter_en == '1';
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    // Event counting is allowed unless it is prohibited by any rule below
    prohibited = FALSE;

    // Event counting in Secure state is permitted as EL3 is not implemented
    // Event counting at EL2 is prohibited if all of:
    // * FEAT_PMUv3p1 is implemented
    // * PMNx is not reserved for EL2
    // * EL2 is using AArch64 and MDCR_EL2.HPMD == 1, or EL2 is using AArch32 and HDCR.HPMD == 1
    if (!prohibited && PSTATE.EL == EL2 && IsFeatureImplemented(FEAT_PMUv3p1) &&
          counter_range == PMUCounterRange_R1) then
        hpmd = MDCR_EL2.HPMD;
        prohibited = hpmd == '1';

    // The IMPLEMENTATION DEFINED authentication interface might override software
    if prohibited && !IsFeatureImplemented(FEAT_Debugv8p2) then
        prohibited = !ExternalSecureNoninvasiveDebugEnabled();

    // PMCR_EL0.DP or PMCR.DP disables the cycle counter when event counting is prohibited
    if prohibited && idx == CYCLE_COUNTER_ID then
        dp = PMCR_EL0.DP;
        enabled = enabled && dp == '0';
        // Otherwise whether event counting is prohibited does not affect the cycle counter
        prohibited = FALSE;

    // Event counting can be filtered by the {P, U, NSK, NSU, NSH} bits
    bits(32) filter;
    case idx of
        when CYCLE_COUNTER_ID
            filter = PMCCFILTR_EL0<31:0>;
        otherwise
            filter = PMEVTYPER_EL0[idx]<31:0>;

    p   = filter<31>;
    u   = filter<30>;
    nsh = if HaveEL(EL2) then filter<27> else '0';

    case PSTATE.EL of
        when EL0
            filtered = u == '1';
        when EL1
            filtered = p == '1';
        when EL2
            filtered = nsh == '0';

    return !debug && enabled && !prohibited && !filtered;
```

G1.2.1.77 EffectiveHPMN

```
// EffectiveHPMN()
// ===============
// Returns the Effective value of MDCR_EL2.HPMN.

bits(5) EffectiveHPMN()
    constant integer counters = NUM_PMU_COUNTERS;
    bits(5) hpmn_bits;

    if HaveEL(EL2) then     // Software can reserve some event counters for EL2
        hpmn_bits = MDCR_EL2.HPMN;

        if UInt(hpmn_bits) > counters || IsZero(hpmn_bits) then
            (-, hpmn_bits) = ConstrainUnpredictableBits(Unpredictable_RES_HPMN, 5);
    else
        hpmn_bits = counters<4:0>;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    return hpmn_bits;
```

G1.2.1.78 GetNumEventCountersAccessible

```
// GetNumEventCountersAccessible()
// ===============================
// Return the number of event counters that can be accessed at the current Exception level.

integer GetNumEventCountersAccessible()
    integer n;

    // Software can reserve some counters for EL2
    if PSTATE.EL IN {EL1, EL0} && EL2Enabled() then
        n = UInt(EffectiveHPMN());
    else
        n = NUM_PMU_COUNTERS;

    return n;
```

G1.2.1.79 GetNumEventCountersSelfHosted

```
// GetNumEventCountersSelfHosted()
// ===============================
// Return the number of event counters that can be accessed by the Self-hosted software.

integer GetNumEventCountersSelfHosted()
    return NUM_PMU_COUNTERS;
```

G1.2.1.80 GetPMUAccessMask

```
// GetPMUAccessMask()
// ==================
// Return a mask of the PMU counters accessible at the current Exception level

bits(64) GetPMUAccessMask()
    bits(64) mask = Zeros(64);

    // PMCCNTR_EL0 is always implemented and accessible
    mask<CYCLE_COUNTER_ID> = '1';

    // PMEVCNTR<n>_EL0
    constant integer counters = GetNumEventCountersAccessible();
    if counters > 0 then
        mask<counters-1:0> = Ones(counters);

    return mask;
```

G1.2.1.81 GetPMUCounterRange

```
// GetPMUCounterRange()
// ====================
// Returns the range that a counter is currently in.

PMUCounterRange GetPMUCounterRange(integer n)
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    constant integer counters = NUM_PMU_COUNTERS;
    constant integer hpmn = UInt(EffectiveHPMN());

    if n < hpmn then
        return PMUCounterRange_R1;
    elsif n < counters then
        return PMUCounterRange_R2;
    elsif n == CYCLE_COUNTER_ID then
        return PMUCounterRange_R1;
    else
        Unreachable();
```

G1.2.1.82 GetPMUReadMask

```
// GetPMUReadMask()
// ================
// Return a mask of the PMU counters that can be read at the current
// Exception level.
// This mask masks reads from PMCNTENSET_EL0, PMCNTENCLR_EL0, PMINTENSET_EL1,
// PMINTENCLR_EL1, PMOVSSET_EL0, and PMOVSCLR_EL0.

bits(64) GetPMUReadMask()
    bits(64) mask = GetPMUAccessMask();
    return mask;
```

G1.2.1.83 GetPMUWriteMask

```
// GetPMUWriteMask()
// =================
// Return a mask of the PMU counters writable at the current Exception level.
// This mask masks writes to PMCNTENSET_EL0, PMCNTENCLR_EL0, PMINTENSET_EL1,
// PMINTENCLR_EL1, PMOVSSET_EL0, and PMOVSCLR_EL0.

bits(64) GetPMUWriteMask()
    bits(64) mask = GetPMUAccessMask();

    return mask;
```

G1.2.1.84 HasElapsed64Cycles

```
// HasElapsed64Cycles()
// ====================
// Returns TRUE if 64 cycles have elapsed between the last count, and FALSE otherwise.

boolean HasElapsed64Cycles();
```

G1.2.1.85 PMUCounterRange

```
// PMUCounterRange
// ===============
// Enumerates the ranges to which an event counter belongs to.

enumeration PMUCounterRange {
    PMUCounterRange_R1,
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    PMUCounterRange_R2
    };
```

G1.2.1.86 PMUEvent

```
// PMUEvent()
// ==========
// Generate a PMU event. By default, increment by 1.

PMUEvent(bits(16) pmuevent)
    PMUEvent(pmuevent, 1);

// PMUEvent()
// ==========
// Accumulate a PMU Event.

PMUEvent(bits(16) pmuevent, integer increment)
    constant integer counters = NUM_PMU_COUNTERS;
    if counters != 0 then
        for idx = 0 to counters - 1
            PMUEvent(pmuevent, increment, idx);

// PMUEvent()
// ==========
// Accumulate a PMU Event for a specific event counter.

PMUEvent(bits(16) pmuevent, integer increment, integer idx)
    if !IsFeatureImplemented(FEAT_PMUv3) then
        return;
    if PMEVTYPER_EL0[idx].evtCount == pmuevent then
        PMUEventAccumulator[idx] = PMUEventAccumulator[idx] + increment;
```

G1.2.1.87 PMUOverflowCondition

```
// PMUOverflowCondition()
// ======================
// Enumerates the reasons for which the PMU overflow condition is evaluated.

enumeration PMUOverflowCondition {
    PMUOverflowCondition_IRQ,
    };
```

G1.2.1.88 PMUSwIncrement

```
// PMUSwIncrement()
// ================
// Generate PMU Events on a write to PMSWINC

PMUSwIncrement(bits(64) sw_incr_in)

    bits(64) sw_incr = sw_incr_in;
    sw_incr = sw_incr AND GetPMUAccessMask();
    for idx = 0 to 30
        if sw_incr<idx> == '1' then
            PMUEvent(PMU_EVENT_SW_INCR, 1, idx);

    return;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.1.89 ZeroCycleCounter

```
// ZeroCycleCounter()
// ==================
// Called on a write to PMCR_EL0 or PMCR that writes '1' to PMCR_EL0.C or PMCR.C.

ZeroCycleCounter()
    bits(64) mask = Zeros(64);
    mask<CYCLE_COUNTER_ID> = '1';
    ZeroPMUCounters(mask);
```

G1.2.1.90 ZeroPMUCounters

```
// ZeroPMUCounters()
// =================
// Zero set of counters specified by the mask in 'val'.

ZeroPMUCounters(bits(64) val)
    constant bits(64) masked_val = val AND GetPMUWriteMask();

    for idx = 0 to 63
        if masked_val<idx> == '1' then
            case idx of
                when CYCLE_COUNTER_ID
                    PMCCNTR_EL0 = Zeros(64);
                otherwise
                    PMEVCNTR_EL0[idx] = Zeros(64);

    return;
```

G1.2.1.91 CreatePCSample

```
// CreatePCSample()
// ================

CreatePCSample()
    // In a simple sequential execution of the program, CreatePCSample is executed each time the PE
    // executes an instruction that can be sampled. An implementation is not constrained such that
    // reads of EDPCSRlo return the current values of PC, etc.
    if PCSRSuspended() then return;
    pc_sample.valid = ExternalNoninvasiveDebugAllowed() && !Halted();
    pc_sample.pc = ThisInstrAddr(64);
    pc_sample.el = PSTATE.EL;
    pc_sample.rw = if UsingAArch32() then '0' else '1';
    pc_sample.ss = CurrentSecurityState();
    pc_sample.contextidr = CONTEXTIDR_EL1<31:0>;
    pc_sample.has_el2 = PSTATE.EL != EL3 && EL2Enabled();

    if pc_sample.has_el2 then
        pc_sample.vmid = VMID[];
        pc_sample.contextidr_el2 = CONTEXTIDR_EL2<31:0>;
        pc_sample.el0h = FALSE;
    return;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.1.92 PCSRSuspended

```
// PCSRSuspended()
// ===============
// Returns TRUE if PC Sample-based Profiling is suspended, and FALSE otherwise.

boolean PCSRSuspended()
    return boolean IMPLEMENTATION_DEFINED "PCSR is suspended";
```

G1.2.1.93 PCSample

```
PCSample pc_sample;

// PCSample
// ========

type PCSample is (
    boolean valid,
    bits(64) pc,
    bits(2) el,
    bit rw,
    SecurityState ss,
    boolean has_el2,
    bits(32) contextidr,
    bits(32) contextidr_el2,
    boolean el0h,
    bits(16) vmid
)
```

G1.2.1.94 Read_PMPCSR

```
// Read_PMPCSR()
// =============

bits(64) Read_PMPCSR(boolean memory_mapped)
    // The Software lock is OPTIONAL.
    update = !memory_mapped || PMLSR.SLK == '0';        // Software locked: no side-effects

    if pc_sample.valid then
        if update then SetPCSample();
        return PMPCSR;
    else
        if update then SetPCSRUnknown();
        return (bits(32) UNKNOWN : Ones(32));
```

G1.2.1.95 SetPCSRUnknown

```
// SetPCSRUnknown()
// ================
// Sets the PC sample registers to UNKNOWN values because PC sampling
// is prohibited.

SetPCSRUnknown()
    PMPCSR<31:0>  = Ones(32);
    PMPCSR<55:32> = bits(24) UNKNOWN;
    PMPCSR.EL     = bits(2) UNKNOWN;
    PMPCSR.NS     = bit UNKNOWN;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if IsFeatureImplemented(FEAT_PMUv3_EXT32) then
        PMCID1SR      = bits(32) UNKNOWN;
        PMCID2SR      = bits(32) UNKNOWN;
        PMVIDSR.VMID  = bits(16) UNKNOWN;

    return;
```

G1.2.1.96 SetPCSample

```
// SetPCSample()
// =============
// Sets the PC sample registers to the appropriate sample values.

SetPCSample()
    PMPCSR<31:0> = pc_sample.pc<31:0>;
    PMPCSR<55:32> = (if pc_sample.rw == '0' then Zeros(24) else pc_sample.pc<55:32>);
    PMPCSR.EL = pc_sample.el;
    PMPCSR.NS = (if pc_sample.ss == SS_Secure then '0' else '1');
    constant bits(32) contextidr_el2 = (if pc_sample.has_el2 then
                                        pc_sample.contextidr_el2 else bits(32) UNKNOWN);
    bits(16) vmid = bits(16) UNKNOWN;
    if pc_sample.has_el2 && pc_sample.el IN {EL1,EL0} && !pc_sample.el0h then
        vmid = pc_sample.vmid;
    if IsFeatureImplemented(FEAT_PMUv3_EXT32) then
        PMCID1SR = pc_sample.contextidr;
        PMCID2SR = contextidr_el2;
        PMVIDSR.VMID = vmid;

    return;
```

G1.2.1.97 CheckSoftwareStep

```
// CheckSoftwareStep()
// ===================
// Take a Software Step exception if in the active-pending state

CheckSoftwareStep()

    // Other self-hosted debug functions will call AArch32.GenerateDebugExceptions() if called from
    // AArch32 state. However, because Software Step is only active when the debug target Exception
    // level is using AArch64, CheckSoftwareStep only calls AArch64.GenerateDebugExceptions().
    step_enabled = (!ELUsingAArch32(DebugTarget()) && AArch64.GenerateDebugExceptions() &&
                    MDSCR_EL1.SS == '1');
    active_pending = step_enabled && PSTATE.SS == '0';   // active-pending
    if active_pending then
        AArch64.SoftwareStepException();
    ShouldAdvanceSS = TRUE;
    return;
```

G1.2.1.98 DebugExceptionReturnSS

```
// DebugExceptionReturnSS()
// ========================
// Returns value to write to PSTATE.SS on an exception return or Debug state exit.

bit DebugExceptionReturnSS(bits(N) spsr)
    assert Halted() || Restarting() ||  PSTATE.EL != EL0;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    boolean enabled_at_source;
    if Restarting() then
        enabled_at_source = FALSE;
    else
        enabled_at_source = AArch64.GenerateDebugExceptions();
    boolean valid;
    bits(2) dest_el;
    if IllegalExceptionReturn(spsr) then
        dest_el = PSTATE.EL;
    else
        (valid, dest_el) = ELFromSPSR(spsr);  assert valid;

    dest_ss = SecurityStateAtEL(dest_el);
    constant bit mask = spsr<9>;
    constant boolean enabled_at_dest = AArch64.GenerateDebugExceptionsFrom(dest_el, dest_ss, mask);

    ELd = DebugTargetFrom(dest_ss);
    bit SS_bit;
    if !ELUsingAArch32(ELd) && MDSCR_EL1.SS == '1' && !enabled_at_source && enabled_at_dest then
        SS_bit = spsr<21>;
    else
        SS_bit = '0';
    return SS_bit;
```

G1.2.1.99 SSAdvance

```
// SSAdvance()
// ===========
// Advance the Software Step state machine.

SSAdvance()

    // A simpler implementation of this function just clears PSTATE.SS to zero regardless of the
    // current Software Step state machine. However, this check is made to illustrate that the
    // PE only needs to consider advancing the state machine from the active-not-pending
    // state.
    if !ShouldAdvanceSS then return;
    target = DebugTarget();
    step_enabled = !ELUsingAArch32(target) && MDSCR_EL1.SS == '1';
    active_not_pending = step_enabled && PSTATE.SS == '1';
    if active_not_pending then PSTATE.SS = '0';
    ShouldAdvanceSS = FALSE;
    return;
```

G1.2.1.100 SoftwareStep_DidNotStep

```
// SoftwareStep_DidNotStep()
// =========================
// Returns TRUE if the previously executed instruction was executed in the
// inactive state, that is, if it was not itself stepped.
// Might return TRUE or FALSE if the previously executed instruction was an ISB
// or ERET executed in the active-not-pending state, or if another exception
// was taken before the Software Step exception.  Returns FALSE otherwise,
// indicating that the previously executed instruction was executed in the
// active-not-pending state, that is, the instruction was stepped.

boolean SoftwareStep_DidNotStep();
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.1.101 SoftwareStep_SteppedEX

```
// SoftwareStep_SteppedEX()
// ========================
// Returns a value that describes the previously executed instruction. The
// result is valid only if SoftwareStep_DidNotStep() returns FALSE.
// Returns TRUE if the instruction was a Load-Exclusive class instruction,
// and FALSE if the instruction was not a Load-Exclusive class instruction.

boolean SoftwareStep_SteppedEX();
```

G1.2.1.102 DataCacheWatchpointSize

```
// DataCacheWatchpointSize()
// =========================
// Return the IMPLEMENTATION DEFINED data cache watchpoint size

integer DataCacheWatchpointSize()
    constant integer size = integer IMPLEMENTATION_DEFINED "Data Cache Invalidate Watchpoint Size";
    assert IsPow2(size) && size >= 2^(UInt(CTR_EL0.DminLine) + 2) && size <= 2048;
    return size;
```

G1.2.1.103 WatchpointInfo

```
// WatchpointInfo
// ==============
// Watchpoint related fields.

type WatchpointInfo is (
    integer  watchpt_num,       // Matching watchpoint number
    boolean  value_match,       // Watchpoint match
    bits(64) vaddress           // Matching Virtual Address
)
```

G1.2.2 shared/exceptions

This section includes the following pseudocode functions:

G1.2.2.1 EffectiveHCR_AMO

```
// EffectiveHCR_AMO()
// ==================
// Return the Effective value of HCR_EL2.AMO.

bit EffectiveHCR_AMO()
    if EffectiveTGE() == '1' then
        return '1';
    elsif EL2Enabled() then
        return HCR_EL2.AMO;
    else
        return '0';
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.2.2 EffectiveHCR_TEA

```
// EffectiveHCR_TEA()
// ==================
// Return the Effective value of HCR_EL2.TEA.

bit EffectiveHCR_TEA()
    if EL2Enabled() && IsFeatureImplemented(FEAT_RAS) then
        return HCR_EL2.TEA;
    else
        return '0';
```

G1.2.2.3 PhysicalSErrorTarget

```
// PhysicalSErrorTarget()
// ======================
// Returns a tuple of whether SError exception can be taken and, if so, the
// target Exception level.

(boolean, bits(2)) PhysicalSErrorTarget()
    if Halted() then
        return (TRUE, bits(2) UNKNOWN);

    constant bit effective_ea = '0';          // EL3 is not implemented
    constant bit effective_amo = EffectiveHCR_AMO();
    constant bit effective_tge = EffectiveTGE();

    // The exception is masked by software.
    boolean masked;
    case PSTATE.EL of
        when EL2
            masked = (effective_ea == '0' &&
                      ((!UsingAArch32() && effective_tge == '0' && effective_amo == '0') ||
                       PSTATE.A == '1'));
        when EL1, EL0
            masked = (effective_ea == '0' && effective_amo == '0' && PSTATE.A == '1');

    // External debug might disable the exception in the current Security state.
    // This is not relevant at EL3.
    constant boolean intdis = PSTATE.EL != EL3 && ExternalDebugInterruptsDisabled(EL1);

    bits(2) target_el = bits(2) UNKNOWN;

    if EL2Enabled() && effective_amo == '1' && !intdis && PSTATE.EL IN {EL0, EL1} then
        target_el = EL2;
        masked = FALSE;

    elsif PSTATE.EL == EL2 then
        if !masked then target_el = EL2;

    else
        assert (PSTATE.EL == EL1 ||
                PSTATE.EL == EL0);
        if !masked then target_el = EL1;

    // External debug might disable the exception for the target Exception level.
    if !masked && ExternalDebugInterruptsDisabled(target_el) then
        masked = TRUE;
        target_el = bits(2) UNKNOWN;

    return (masked, target_el);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.2.4 SyncExternalAbortTarget

```
// SyncExternalAbortTarget()
// =========================
// Returns the target Exception level for a Synchronous External Data or
// Instruction or Prefetch Abort.

bits(2) SyncExternalAbortTarget(FaultRecord fault)
    constant bit effective_tea = EffectiveHCR_TEA();
    constant bit effective_tge = EffectiveTGE();

    bits(2) target_el;

    if (EL2Enabled() && PSTATE.EL IN {EL1, EL0} &&
           (effective_tea == '1' || IsSecondStage(fault) ||
            (PSTATE.EL == EL0 && effective_tge == '1'))) then
        target_el = EL2;

    else
        assert PSTATE.EL != EL3 || ELUsingAArch32(EL3);
        target_el = (if PSTATE.EL == EL2 then EL2 else EL1);

    return target_el;
```

G1.2.2.5 ConditionSyndrome

```
// ConditionSyndrome()
// ===================
// Return CV and COND fields of instruction syndrome

bits(5) ConditionSyndrome()

    bits(5) syndrome;

    syndrome<4> = '1';
    syndrome<3:0> = '1110';
    return syndrome;
```

G1.2.2.6 Exception

```
// Exception
// =========
// Classes of exception.

enumeration Exception {
        Exception_Uncategorized,        // Uncategorized or unknown reason
        Exception_WFxTrap,              // Trapped WFI or WFE instruction
        Exception_CP15RTTrap,           // Trapped AArch32 MCR or MRC access, coproc=0b111
        Exception_CP15RRTTrap,          // Trapped AArch32 MCRR or MRRC access, coproc=0b1111
        Exception_CP14RTTrap,           // Trapped AArch32 MCR or MRC access, coproc=0b1110
        Exception_CP14DTTrap,           // Trapped AArch32 LDC or STC access, coproc=0b1110
        Exception_CP14RRTTrap,          // Trapped AArch32 MRRC access, coproc=0b1110
        Exception_AdvSIMDFPAccessTrap,  // HCPTR-trapped access to SIMD or FP
        Exception_FPIDTrap,             // Trapped access to SIMD or FP ID register
        Exception_LDST64BTrap,          // Trapped access to ST64B and LD64B
        // Trapped BXJ instruction not supported in Armv8
        Exception_PACTrap,               // Trapped invalid PAC use
        Exception_IllegalState,          // Illegal Execution state
        Exception_SupervisorCall,        // Supervisor Call
        Exception_HypervisorCall,        // Hypervisor Call
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Exception_MonitorCall,          // Monitor Call or Trapped SMC instruction
        Exception_SystemRegisterTrap,   // Trapped MRS or MSR System register access
        Exception_InstructionAbort,     // Instruction Abort or Prefetch Abort
        Exception_PCAlignment,          // PC alignment fault
        Exception_DataAbort,            // Data Abort
        Exception_PACFail,              // PAC Authentication failure
        Exception_SPAlignment,          // SP alignment fault
        Exception_FPTrappedException,   // IEEE trapped FP exception
        Exception_SError,               // SError interrupt
        Exception_Breakpoint,           // (Hardware) Breakpoint
        Exception_SoftwareStep,         // Software Step
        Exception_Watchpoint,           // Watchpoint
        Exception_SoftwareBreakpoint,   // Software Breakpoint Instruction
        Exception_VectorCatch,          // AArch32 Vector Catch
        Exception_IRQ,                  // IRQ interrupt
        Exception_FIQ};                // FIQ interrupt
```

G1.2.2.7 ExceptionRecord

```
// ExceptionRecord
// ===============

type ExceptionRecord is (
    Exception   exceptype,          // Exception class
    IssType     syndrome,           // Syndrome record
    bits(64)    vaddress,           // Virtual fault address
    boolean     ipavalid,           // Validity of Intermediate Physical fault address
    bit         NS,                 // Intermediate Physical fault address space
    bits(52)    ipaddress,          // Intermediate Physical fault address
    boolean     trappedsyscallinst) // Trapped SVC or SMC instruction
```

G1.2.2.8 ExceptionSyndrome

```
// ExceptionSyndrome()
// ===================
// Return a blank exception syndrome record for an exception of the given type.

ExceptionRecord ExceptionSyndrome(Exception exceptype)

    ExceptionRecord r;

    r.exceptype = exceptype;

    // Initialize all other fields
    r.syndrome.iss    = Zeros(25);
    r.syndrome.iss2   = Zeros(24);
    r.vaddress        = Zeros(64);
    r.ipavalid        = FALSE;
    r.NS              = '0';
    r.ipaddress       = Zeros(52);
    r.trappedsyscallinst = FALSE;
    return r;
```

G1.2.2.9 Undefined

```
// Undefined()
// ===========
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
Undefined()
    AArch64.Undefined();
```

G1.2.3 shared/functions

This section includes the following pseudocode functions:

G1.2.3.1 EncodeLDFSC

```
// EncodeLDFSC()
// =============
// Function that gives the Long-descriptor FSC code for types of Fault

bits(6) EncodeLDFSC(Fault statuscode, integer level)
    bits(6) result;

    case statuscode of
        when Fault_AddressSize         result = '0000':level<1:0>; assert level IN {0,1,2,3};
        when Fault_AccessFlag          result = '0010':level<1:0>; assert level IN {1,2,3};
        when Fault_Permission          result = '0011':level<1:0>; assert level IN {0,1,2,3};
        when Fault_Translation         result = '0001':level<1:0>; assert level IN {0,1,2,3};
        when Fault_SyncExternal        result = '010000';
        when Fault_SyncExternalOnWalk  result = '0101':level<1:0>; assert level IN {0,1,2,3};
        when Fault_SyncParity          result = '011000';
        when Fault_SyncParityOnWalk    result = '0111':level<1:0>; assert level IN {0,1,2,3};
        when Fault_AsyncParity         result = '011001';
        when Fault_AsyncExternal       result = '010001'; assert UsingAArch32();
        when Fault_Alignment           result = '100001';
        when Fault_Debug               result = '100010';
        when Fault_TLBConflict         result = '110000';
        when Fault_HWUpdateAccessFlag  result = '110001';
        when Fault_Lockdown            result = '110100';  // IMPLEMENTATION DEFINED
        when Fault_Exclusive           result = '110101';  // IMPLEMENTATION DEFINED
        otherwise                      Unreachable();

    return result;
```

G1.2.3.2 IPAValid

```
// IPAValid()
// ==========
// Return TRUE if the IPA is reported for the abort

boolean IPAValid(FaultRecord fault)
    assert fault.statuscode != Fault_None;

    if fault.s2fs1walk then
        return fault.statuscode IN {
            Fault_AccessFlag,
            Fault_Permission,
            Fault_Translation,
            Fault_AddressSize
        };
    elsif fault.secondstage then
        return fault.statuscode IN {
            Fault_AccessFlag,
            Fault_Translation,
            Fault_AddressSize
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        };
    else
        return FALSE;
```

G1.2.3.3 IsAsyncAbort

```
// IsAsyncAbort()
// ==============
// Returns TRUE if the abort currently being processed is an asynchronous abort, and FALSE
// otherwise.

boolean IsAsyncAbort(Fault statuscode)
    assert statuscode != Fault_None;

    return (statuscode IN {Fault_AsyncExternal, Fault_AsyncParity});

// IsAsyncAbort()
// ==============

boolean IsAsyncAbort(FaultRecord fault)
    return IsAsyncAbort(fault.statuscode);
```

G1.2.3.4 IsDebugException

```
// IsDebugException()
// ==================

boolean IsDebugException(FaultRecord fault)
    assert fault.statuscode != Fault_None;
    return fault.statuscode == Fault_Debug;
```

G1.2.3.5 IsExternalAbort

```
// IsExternalAbort()
// =================
// Returns TRUE if the abort currently being processed is an External abort and FALSE otherwise.

boolean IsExternalAbort(Fault statuscode)
    assert statuscode != Fault_None;

    return (statuscode IN {
        Fault_SyncExternal,
        Fault_SyncParity,
        Fault_SyncExternalOnWalk,
        Fault_SyncParityOnWalk,
        Fault_AsyncExternal,
        Fault_AsyncParity
    });

// IsExternalAbort()
// =================

boolean IsExternalAbort(FaultRecord fault)
    return IsExternalAbort(fault.statuscode);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.3.6 IsExternalAbortOnWalk

```
// IsExternalAbortOnWalk()
// =======================

boolean IsExternalAbortOnWalk(FaultRecord fault)
    assert fault.statuscode != Fault_None;
    return fault.statuscode IN {Fault_SyncExternalOnWalk, Fault_SyncParityOnWalk};
```

G1.2.3.7 IsExternalSyncAbort

```
// IsExternalSyncAbort()
// =====================
// Returns TRUE if the abort currently being processed is an external
// synchronous abort and FALSE otherwise.

boolean IsExternalSyncAbort(Fault statuscode)
    assert statuscode != Fault_None;
    if IsFeatureImplemented(FEAT_RAS) then
        assert ! statuscode IN {Fault_SyncParity, Fault_SyncParityOnWalk};

    return (statuscode IN {
        Fault_SyncExternal,
        Fault_SyncParity,
        Fault_SyncExternalOnWalk,
        Fault_SyncParityOnWalk
    });

// IsExternalSyncAbort()
// =====================

boolean IsExternalSyncAbort(FaultRecord fault)
    return IsExternalSyncAbort(fault.statuscode);
```

G1.2.3.8 IsFault

```
// IsFault()
// =========
// Return TRUE if a fault is associated with an address descriptor

boolean IsFault(AddressDescriptor addrdesc)
    return addrdesc.fault.statuscode != Fault_None;

// IsFault()
// =========
// Return TRUE if a fault is associated with a memory access.

boolean IsFault(Fault fault)
    return fault != Fault_None;

// IsFault()
// =========
// Return TRUE if a fault is associated with status returned by memory.

boolean IsFault(PhysMemRetStatus retstatus)
    return retstatus.statuscode != Fault_None;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.3.9 IsSErrorInterrupt

```
// IsSErrorInterrupt()
// ===================
// Returns TRUE if the abort currently being processed is an SError interrupt, and FALSE
// otherwise.

boolean IsSErrorInterrupt(Fault statuscode)
    assert statuscode != Fault_None;

    return (statuscode IN {Fault_AsyncExternal, Fault_AsyncParity});

// IsSErrorInterrupt()
// ===================

boolean IsSErrorInterrupt(FaultRecord fault)
    return IsSErrorInterrupt(fault.statuscode);

// Add a specific type of return value for FaultSyndrome
type IssType is (
    bits(25) iss,
    bits(24) iss2
)
```

G1.2.3.10 IsSecondStage

```
// IsSecondStage()
// ===============

boolean IsSecondStage(FaultRecord fault)
    assert fault.statuscode != Fault_None;

    return fault.secondstage;
```

G1.2.3.11 LSInstructionSyndrome

```
// LSInstructionSyndrome()
// =======================
// Returns the extended syndrome information for a second stage fault.
//  <10>  - Syndrome valid bit. The syndrome is valid only for certain types of access instruction.
//  <9:8> - Access size.
//  <7>   - Sign extended (for loads).
//  <6:2> - Transfer register.
//  <1>   - Transfer register is 64-bit.
//  <0>   - Instruction has acquire/release semantics.

bits(11) LSInstructionSyndrome();
```

G1.2.3.12 CACHE_OP

```
// CACHE_OP()
// ==========
// Performs Cache maintenance operations as per CacheRecord.

CACHE_OP(CacheRecord cache)
    IMPLEMENTATION_DEFINED;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.3.13 CPASAtPAS

```
// CPASAtPAS()
// ===========
// Get cache PA space for given PA space.

CachePASpace CPASAtPAS(PASpace pas)
    case pas of
        when PAS_NonSecure
            return CPAS_NonSecure;
        when PAS_Secure
            return CPAS_Secure;
        otherwise
            Unreachable();
```

G1.2.3.14 CPASAtSecurityState

```
// CPASAtSecurityState()
// =====================
// Get cache PA space for given security state.

CachePASpace CPASAtSecurityState(SecurityState ss)
    case ss of
        when SS_NonSecure
            return CPAS_NonSecure;
        when SS_Secure
            return CPAS_SecureNonSecure;
```

G1.2.3.15 CacheRecord

```
// CacheRecord
// ===========
// Details related to a cache operation.

type CacheRecord is (
    AccessType       acctype,           // Access type
    CacheOp          cacheop,           // Cache operation
    CacheOpScope     opscope,           // Cache operation type
    CacheType        cachetype,         // Cache type
    bits(64)         regval,
    FullAddress      paddress,
    bits(64)         vaddress,          // For VA operations
    integer          setnum,            // For SW operations
    integer          waynum,            // For SW operations
    integer          level,             // For SW operations
    Shareability     shareability,
    boolean          is_vmid_valid,     // is vmid valid for current context
    bits(16)         vmid,
    boolean          is_asid_valid,     // is asid valid for current context
    bits(16)         asid,
    SecurityState    security,
    // For cache operations to full cache or by setnum/waynum
    // For operations by address, PA space in paddress
    CachePASpace     cpas
)
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.3.16 DecodeSW

```
// DecodeSW()
// ==========
// Decode input value into setnum, waynum and level for SW instructions.

(integer, integer, integer) DecodeSW(bits(64) regval, CacheType cachetype)
    constant integer level = UInt(regval<3:1>);
    (numsets, associativity, linesize) = GetCacheInfo(level, cachetype);
    // For the given level and cachetype, get the number of sets, associativity and
    // cache line size in terms of actual bytes.

    constant integer waybits  = CeilLog2(associativity);
    constant integer setbits  = CeilLog2(numsets);
    constant integer linebits = Log2(linesize);

    constant integer waynum = if associativity == 1 then 0 else UInt(regval<31:32-waybits>);
    constant integer setnum = if numsets == 1 then 0 else UInt(regval<linebits +: setbits>);

    return (setnum, waynum, level);
```

G1.2.3.17 GetCacheInfo

```
// GetCacheInfo()
// ==============
// Returns numsets, assosciativity & linesize in terms of actual bytes.

(integer, integer, integer) GetCacheInfo(integer level, CacheType cachetype);
```

G1.2.3.18 ASR

```
// ASR()
// =====

bits(N) ASR(bits(N) x, integer shift)
    assert shift >= 0;
    bits(N) result;
    if shift == 0 then
        result = x;
    else
        (result, -) = ASR_C(x, shift);
    return result;
```

G1.2.3.19 ASR_C

```
// ASR_C()
// =======

(bits(N), bit) ASR_C(bits(N) x, integer shift)
    assert shift > 0 && shift < 256;
    extended_x = SignExtend(x, shift+N);
    result = extended_x<(shift+N)-1:shift>;
    carry_out = extended_x<shift-1>;
    return (result, carry_out);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.3.20 Abs

```
// Abs()
// =====

integer Abs(integer x)
    return if x >= 0 then x else -x;

// Abs()
// =====

real Abs(real x)
    return if x >= 0.0 then x else -x;
```

G1.2.3.21 Align

```
// Align()
// =======

integer Align(integer x, integer y)
    return y * (x DIV y);

// Align()
// =======

bits(N) Align(bits(N) x, integer y)
    return Align(UInt(x), y)<N-1:0>;
```

G1.2.3.22 BitCount

```
// BitCount()
// ==========

integer BitCount(bits(N) x)
    integer result = 0;
    for i = 0 to N-1
        if x<i> == '1' then
            result = result + 1;
    return result;
```

G1.2.3.23 CeilLog2

```
// CeilLog2()
// ==========
// For a positive integer X, return the Log2() of X, rounded up to the next integer

integer CeilLog2(integer x)
    assert x != 0;
    return Log2(CeilPow2(x));
```

G1.2.3.24 CeilPow2

```
// CeilPow2()
// ==========
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// For a positive integer X, return the smallest power of 2 >= X

integer CeilPow2(integer x)
    if x == 0 then return 0;
    if x == 1 then return 1;
    return FloorPow2(x - 1) * 2;
```

G1.2.3.25 CountLeadingSignBits

```
// CountLeadingSignBits()
// ======================

integer CountLeadingSignBits(bits(N) x)
    return CountLeadingZeroBits(x<N-1:1> EOR x<N-2:0>);
```

G1.2.3.26 CountLeadingZeroBits

```
// CountLeadingZeroBits()
// ======================

integer CountLeadingZeroBits(bits(N) x)
    return N - (HighestSetBit(x) + 1);
```

G1.2.3.27 Elem

```
// Elem - getter
// =============

bits(size) Elem[bits(N) vector, integer e, integer size]
    assert e >= 0 && (e+1)*size <= N;
    constant integer ebase = e * size;
    return vector<ebase + size - 1 : ebase>;

// Elem - setter
// =============

Elem[bits(N) &vector, integer e, integer size] = bits(size) value
    assert e >= 0 && (e+1)*size <= N;
    constant integer ebase = e * size;
    vector<ebase + size - 1 : ebase> = value;
    return;
```

G1.2.3.28 Extend

```
// Extend()
// ========

bits(N) Extend(bits(M) x, integer N, boolean unsigned)
    return if unsigned then ZeroExtend(x, N) else SignExtend(x, N);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.3.29 FloorPow2

```
// FloorPow2()
// ===========
// For a positive integer X, return the largest power of 2 <= X

integer FloorPow2(integer x)
    assert x >= 0;
    integer n = 1;
    if x == 0 then return 0;
    while x >= 2^n do
        n = n + 1;
    return 2^(n - 1);
```

G1.2.3.30 HighestSetBit

```
// HighestSetBit()
// ===============

integer HighestSetBit(bits(N) x)
    for i = N-1 downto 0
        if x<i> == '1' then return i;
    return -1;
```

G1.2.3.31 HighestSetBitNZ

```
// HighestSetBitNZ
// ===============
// Position of the highest 1 bit in a bitvector.
// Asserts if the bitvector is entirely zero.

integer HighestSetBitNZ(bits(N) x)
    assert !IsZero(x);
    return HighestSetBit(x);
```

G1.2.3.32 IsAligned

```
// IsAligned()
// ===========

boolean IsAligned(bits(N) x, integer y)
    return x == Align(x, y);
```

G1.2.3.33 IsAlignedP2

```
// IsAlignedP2()
// =============

boolean IsAlignedP2(bits(N) x, integer p2)
    if N == 0 || p2 == 0 then return TRUE;
    return IsZero(x<p2-1:0>);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.3.34 IsEven

```
// IsEven()
// ========

boolean IsEven(integer val)
    return val MOD 2 == 0;
```

G1.2.3.35 IsOdd

```
// IsOdd()
// =======

boolean IsOdd(integer val)
    return val MOD 2 == 1;
```

G1.2.3.36 IsOnes

```
// IsOnes()
// ========

boolean IsOnes(bits(N) x)
    return x == Ones(N);
```

G1.2.3.37 IsPow2

```
// IsPow2()
// ========
// Return TRUE if integer X is positive and a power of 2. Otherwise,
// return FALSE.

boolean IsPow2(integer x)
    if x <= 0 then return FALSE;
    return FloorPow2(x) == CeilPow2(x);
```

G1.2.3.38 IsZero

```
// IsZero()
// ========

boolean IsZero(bits(N) x)
    return x == Zeros(N);
```

G1.2.3.39 IsZeroBit

```
// IsZeroBit()
// ===========

bit IsZeroBit(bits(N) x)
    return if IsZero(x) then '1' else '0';
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.3.40 LSL

```
// LSL()
// =====

bits(N) LSL(bits(N) x, integer shift)
    assert shift >= 0;
    bits(N) result;
    if shift == 0 then
        result = x;
    else
        (result, -) = LSL_C(x, shift);
    return result;
```

G1.2.3.41 LSL_C

```
// LSL_C()
// =======

(bits(N), bit) LSL_C(bits(N) x, integer shift)
    assert shift > 0 && shift < 256;
    extended_x = x : Zeros(shift);
    result = extended_x<N-1:0>;
    carry_out = extended_x<N>;
    return (result, carry_out);
```

G1.2.3.42 LSR

```
// LSR()
// =====

bits(N) LSR(bits(N) x, integer shift)
    assert shift >= 0;
    bits(N) result;
    if shift == 0 then
        result = x;
    else
        (result, -) = LSR_C(x, shift);
    return result;
```

G1.2.3.43 LSR_C

```
// LSR_C()
// =======

(bits(N), bit) LSR_C(bits(N) x, integer shift)
    assert shift > 0 && shift < 256;
    extended_x = ZeroExtend(x, shift+N);
    result = extended_x<(shift+N)-1:shift>;
    carry_out = extended_x<shift-1>;
    return (result, carry_out);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.3.44 LowestSetBit

```
// LowestSetBit()
// ==============

integer LowestSetBit(bits(N) x)
    for i = 0 to N-1
        if x<i> == '1' then return i;
    return N;
```

G1.2.3.45 LowestSetBitNZ

```
// LowestSetBitNZ
// ==============
// Position of the lowest 1 bit in a bitvector.
// Asserts if the bit-vector is entirely zero.

integer LowestSetBitNZ(bits(N) x)
    assert !IsZero(x);
    return LowestSetBit(x);
```

G1.2.3.46 Max

```
// Max()
// =====

integer Max(integer a, integer b)
    return if a >= b then a else b;

// Max()
// =====

real Max(real a, real b)
    return if a >= b then a else b;
```

G1.2.3.47 Min

```
// Min()
// =====

integer Min(integer a, integer b)
    return if a <= b then a else b;

// Min()
// =====

real Min(real a, real b)
    return if a <= b then a else b;
```

G1.2.3.48 NormalizeReal

```
// NormalizeReal
// =============
// Normalizes x to the form 1.xxx... x 2^y and returns (mantissa, exponent)
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
(real, integer) NormalizeReal(real x)
    real mantissa = x;
    integer exponent = 0;
    while mantissa < 1.0 do
        mantissa = mantissa * 2.0;  exponent = exponent - 1;
    while mantissa >= 2.0 do
        mantissa = mantissa / 2.0;  exponent = exponent + 1;
    return (mantissa, exponent);
```

G1.2.3.49 Ones

```
// Ones()
// ======

bits(N) Ones(integer N)
    return Replicate('1',N);
```

G1.2.3.50 ROL

```
// ROL()
// =====

bits(N) ROL(bits(N) x, integer shift)
    assert shift >= 0 && shift <= N;
    if (shift == 0) then
        return x;
    return ROR(x, N-shift);
```

G1.2.3.51 ROR

```
// ROR()
// =====

bits(N) ROR(bits(N) x, integer shift)
    assert shift >= 0;
    bits(N) result;
    if shift == 0 then
        result = x;
    else
        (result, -) = ROR_C(x, shift);
    return result;
```

G1.2.3.52 ROR_C

```
// ROR_C()
// =======

(bits(N), bit) ROR_C(bits(N) x, integer shift)
    assert shift != 0 && shift < 256;
    m = shift MOD N;
    result = LSR(x,m) OR LSL(x,N-m);
    carry_out = result<N-1>;
    return (result, carry_out);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.3.53 RShr

```
// RShr()
// ======
// Shift integer value right with rounding

integer RShr(integer value, integer shift, boolean round)
    assert shift > 0;
    if round then
        return (value + (1 << (shift - 1))) >> shift;
    else
        return value >> shift;
```

G1.2.3.54 Replicate

```
// Replicate()
// ===========

bits(M*N) Replicate(bits(M) x, integer N);
```

G1.2.3.55 Reverse

```
// Reverse()
// =========
// Reverse subwords of M bits in an N-bit word

bits(N) Reverse(bits(N) word, integer M)
    assert M > 0;
    assert N MOD M == 0;
    bits(N) result;
    constant integer swsize = M;
    constant integer sw = N DIV swsize;
    for s = 0 to sw-1
        Elem[result, (sw - 1) - s, swsize] = Elem[word, s, swsize];
    return result;
```

G1.2.3.56 RoundDown

```
// RoundDown()
// ===========

integer RoundDown(real x);
```

G1.2.3.57 RoundTowardsZero

```
// RoundTowardsZero()
// ==================

integer RoundTowardsZero(real x)
    return if x == 0.0 then 0 else if x >= 0.0 then RoundDown(x) else RoundUp(x);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.3.58 RoundUp

```
// RoundUp()
// =========

integer RoundUp(real x);
```

G1.2.3.59 SInt

```
// SInt()
// ======

integer SInt(bits(N) x)
    result = 0;
    for i = 0 to N-1
        if x<i> == '1' then result = result + 2^i;
    if x<N-1> == '1' then result = result - 2^N;
    return result;
```

G1.2.3.60 SignExtend

```
// SignExtend()
// ============

bits(N) SignExtend(bits(M) x, integer N)
    assert N >= M;
    return Replicate(x<M-1>, N-M) : x;
```

G1.2.3.61 Signal

```
// Signal
// ======
// Available signal types

enumeration Signal {LOW, HIGH};
```

G1.2.3.62 UInt

```
// UInt()
// ======

integer UInt(bits(N) x)
    result = 0;
    for i = 0 to N-1
        if x<i> == '1' then result = result + 2^i;
    return result;
```

G1.2.3.63 ZeroExtend

```
// ZeroExtend()
// ============
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
bits(N) ZeroExtend(bits(M) x, integer N)
    assert N >= M;
    return Zeros(N-M) : x;
```

G1.2.3.64 Zeros

```
// Zeros()
// =======

bits(N) Zeros(integer N)
    return Replicate('0',N);
```

G1.2.3.65 AArch64.CheckTimerConditions

```
// AArch64.CheckTimerConditions()
// ==============================
// Checking timer conditions for all A64 timer registers

AArch64.CheckTimerConditions()
    boolean status;
    bits(64) offset;
    bit imask;
    offset = Zeros(64);
    if CNTP_CTL_EL0.ENABLE == '1' then
        imask = CNTP_CTL_EL0.IMASK;
        status = IsTimerConditionMet(offset, CNTP_CVAL_EL0,
                                     imask, InterruptID_CNTP);
        CNTP_CTL_EL0.ISTATUS = if status then '1' else '0';
    if HaveEL(EL2) && IsFeatureImplemented(FEAT_SEL2) && CNTHPS_CTL_EL2.ENABLE == '1' then
        status = IsTimerConditionMet(Zeros(64), CNTHPS_CVAL_EL2,
                                     CNTHPS_CTL_EL2.IMASK, InterruptID_CNTHPS);
        CNTHPS_CTL_EL2.ISTATUS = if status then '1' else '0';

    if CNTV_CTL_EL0.ENABLE == '1' then
        imask = CNTV_CTL_EL0.IMASK;
        status = IsTimerConditionMet(CNTVOFF_EL2, CNTV_CVAL_EL0,
                                     imask, InterruptID_CNTV);
        CNTV_CTL_EL0.ISTATUS = if status then '1' else '0';

    return;
```

G1.2.3.66 CNTHCTL_EL2_VHE

```
// CNTHCTL_EL2_VHE()
// =================
// In the case where EL2 accesses the CNTKCTL_EL1 register, and the access
// is redirected to CNTHCTL_EL2 as a result of HCR_EL2.E2H being 1,
// then the bits of CNTHCTL_EL2 that are RES0 in CNTKCTL_EL1 are
// treated as being UNKNOWN. This function applies the UNKNOWN behavior.

bits(64) CNTHCTL_EL2_VHE(bits(64) original_value)
    assert PSTATE.EL == EL2;

    bits(64) return_value = original_value;
    return_value<19:18> = bits(2) UNKNOWN;
    return_value<16:10> = bits(7) UNKNOWN;
    return return_value;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.3.67 GenericCounterTick

```
// GenericCounterTick()
// ====================
// Increments PhysicalCount value for every clock tick.

GenericCounterTick()
    bits(64) prev_physical_count;
    if CNTCR.EN == '0' then
        AArch64.CheckTimerConditions();
        return;
    prev_physical_count = PhysicalCountInt();
    if IsFeatureImplemented(FEAT_CNTSC) && CNTCR.SCEN == '1' then
        PhysicalCount = PhysicalCount + ZeroExtend(CNTSCR, 88);
    else
        PhysicalCount<87:24> = PhysicalCount<87:24> + 1;
    AArch64.CheckTimerConditions();
    TestEventCNTP(prev_physical_count, PhysicalCountInt());
    TestEventCNTV(prev_physical_count, PhysicalCountInt());
    return;
```

G1.2.3.68 IsLocalTimeoutEventPending

```
boolean IsLocalTimeoutEventPending;
```

G1.2.3.69 IsTimerConditionMet

```
// IsTimerConditionMet()
// =====================

boolean IsTimerConditionMet(bits(64) offset, bits(64) compare_value,
                            bits(1) imask, InterruptID intid)
    boolean condition_met;
    Signal level;
    condition_met = (UInt(PhysicalCountInt() - offset) - UInt(compare_value)) >= 0;
    level = if condition_met && imask == '0' then HIGH else LOW;
    SetInterruptRequestLevel(intid, level);
    return condition_met;
```

G1.2.3.70 LocalTimeoutVal

```
bits(64) LocalTimeoutVal;           // Value to compare against the Virtual Counter Timer
```

G1.2.3.71 PhysicalCount

```
bits(88) PhysicalCount;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.3.72 SetEventRegister

```
// SetEventRegister()
// ==================
// Sets the Event Register of this PE

SetEventRegister()
    EventRegister = '1';
    return;
```

G1.2.3.73 TestEventCNTP

```
// TestEventCNTP()
// ===============
// Generate Event stream from the physical counter

TestEventCNTP(bits(64) prev_physical_count, bits(64) current_physical_count)
    bits(64) offset;
    bits(1) samplebit, previousbit;
    integer n;
    if CNTHCTL_EL2.EVNTEN == '1' then
        n = UInt(CNTHCTL_EL2.EVNTI);
        offset = Zeros(64);
        samplebit   = (current_physical_count - offset)<n>;
        previousbit = (prev_physical_count - offset)<n>;
        if CNTHCTL_EL2.EVNTDIR == '0' then
            if previousbit == '0' && samplebit == '1' then SetEventRegister();
        else
            if previousbit == '1' && samplebit == '0' then SetEventRegister();
    return;
```

G1.2.3.74 TestEventCNTV

```
// TestEventCNTV()
// ===============
// Generate Event stream from the virtual counter

TestEventCNTV(bits(64) prev_physical_count, bits(64) current_physical_count)
    bits(64) offset;
    bits(1) samplebit, previousbit;
    integer n;
    if CNTKCTL_EL1.EVNTEN == '1' then
        n = UInt(CNTKCTL_EL1.EVNTI);
        offset      = if HaveEL(EL2) then CNTVOFF_EL2 else Zeros(64);

        samplebit   = (current_physical_count - offset)<n>;
        previousbit = (prev_physical_count - offset)<n>;
        if CNTKCTL_EL1.EVNTDIR == '0' then
            if previousbit == '0' && samplebit == '1' then SetEventRegister();
        else
            if previousbit == '1' && samplebit == '0' then SetEventRegister();
    return;
```

G1.2.3.75 BitReverse

```
// BitReverse()
// ============
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
bits(N) BitReverse(bits(N) data)
    bits(N) result;
    for i = 0 to N-1
        result<(N-i)-1> = data<i>;
    return result;
```

G1.2.3.76 Poly32Mod2

```
// Poly32Mod2()
// ============

// Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation

bits(32) Poly32Mod2(bits(N) data_in, bits(32) poly)
    assert N > 32;
    bits(N) data = data_in;
    for i = N-1 downto 32
        if data<i> == '1' then
            data<i-1:0> = data<i-1:0> EOR (poly:Zeros(i-32));
    return data<31:0>;
```

G1.2.3.77 AESInvMixColumns

```
// AESInvMixColumns()
// ==================
// Transformation in the Inverse Cipher that is the inverse of AESMixColumns.

bits(128) AESInvMixColumns(bits (128) op)
    constant bits(4*8) in0 = op< 96+:8> : op< 64+:8> : op< 32+:8> : op<  0+:8>;
    constant bits(4*8) in1 = op<104+:8> : op< 72+:8> : op< 40+:8> : op<  8+:8>;
    constant bits(4*8) in2 = op<112+:8> : op< 80+:8> : op< 48+:8> : op< 16+:8>;
    constant bits(4*8) in3 = op<120+:8> : op< 88+:8> : op< 56+:8> : op< 24+:8>;

    bits(4*8) out0;
    bits(4*8) out1;
    bits(4*8) out2;
    bits(4*8) out3;

    for c = 0 to 3
        out0<c*8+:8> =  (FFmul0E(in0<c*8+:8>) EOR FFmul0B(in1<c*8+:8>) EOR FFmul0D(in2<c*8+:8>) EOR
                         FFmul09(in3<c*8+:8>));
        out1<c*8+:8> =  (FFmul09(in0<c*8+:8>) EOR FFmul0E(in1<c*8+:8>) EOR FFmul0B(in2<c*8+:8>) EOR
                         FFmul0D(in3<c*8+:8>));
        out2<c*8+:8> =  (FFmul0D(in0<c*8+:8>) EOR FFmul09(in1<c*8+:8>) EOR FFmul0E(in2<c*8+:8>) EOR
                         FFmul0B(in3<c*8+:8>));
        out3<c*8+:8> =  (FFmul0B(in0<c*8+:8>) EOR FFmul0D(in1<c*8+:8>) EOR FFmul09(in2<c*8+:8>) EOR
                         FFmul0E(in3<c*8+:8>));

    return (
        out3<3*8+:8> : out2<3*8+:8> : out1<3*8+:8> : out0<3*8+:8> :
        out3<2*8+:8> : out2<2*8+:8> : out1<2*8+:8> : out0<2*8+:8> :
        out3<1*8+:8> : out2<1*8+:8> : out1<1*8+:8> : out0<1*8+:8> :
        out3<0*8+:8> : out2<0*8+:8> : out1<0*8+:8> : out0<0*8+:8>
    );
```

G1.2.3.78 AESInvShiftRows

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// AESInvShiftRows()
// =================
// Transformation in the Inverse Cipher that is inverse of AESShiftRows.

bits(128) AESInvShiftRows(bits(128) op)
    return (
        op< 31: 24> : op< 55: 48> : op< 79: 72> : op<103: 96> :
        op<127:120> : op< 23: 16> : op< 47: 40> : op< 71: 64> :
        op< 95: 88> : op<119:112> : op< 15:  8> : op< 39: 32> :
        op< 63: 56> : op< 87: 80> : op<111:104> : op<  7:  0>
    );
```

G1.2.3.79 AESInvSubBytes

```
// AESInvSubBytes()
// ================
// Transformation in the Inverse Cipher that is the inverse of AESSubBytes.

bits(128) AESInvSubBytes(bits(128) op)
    // Inverse S-box values
    constant bits(16*16*8) GF2_inv = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0x7d0c2155631469e126d677ba7e042b17<127:0> :
        /*E*/ 0x619953833cbbebc8b0f52aae4d3be0a0<127:0> :
        /*D*/ 0xef9cc9939f7ae52d0d4ab519a97f5160<127:0> :
        /*C*/ 0x5fec8027591012b131c7078833a8dd1f<127:0> :
        /*B*/ 0xf45acd78fec0db9a2079d2c64b3e56fc<127:0> :
        /*A*/ 0x1bbe18aa0e62b76f89c5291d711af147<127:0> :
        /*9*/ 0x6edf751ce837f9e28535ade72274ac96<127:0> :
        /*8*/ 0x73e6b4f0cecff297eadc674f4111913a<127:0> :
        /*7*/ 0x6b8a130103bdafc1020f3fca8f1e2cd0<127:0> :
        /*6*/ 0x0645b3b80558e4f70ad3bc8c00abd890<127:0> :
        /*5*/ 0x849d8da75746155edab9edfd5048706c<127:0> :
        /*4*/ 0x92b6655dcc5ca4d41698688664f6f872<127:0> :
        /*3*/ 0x25d18b6d49a25b76b224d92866a12e08<127:0> :
        /*2*/ 0x4ec3fa420b954cee3d23c2a632947b54<127:0> :
        /*1*/ 0xcbe9dec444438e3487ff2f9b8239e37c<127:0> :
        /*0*/ 0xfbd7f3819ea340bf38a53630d56a0952<127:0>
    );
    bits(128) out;
    for i = 0 to 15
        out<i*8+:8> = GF2_inv<UInt(op<i*8+:8>)*8+:8>;
    return out;
```

G1.2.3.80 AESMixColumns

```
// AESMixColumns()
// ===============
// Transformation in the Cipher that takes all of the columns of the
// State and mixes their data (independently of one another) to
// produce new columns.

bits(128) AESMixColumns(bits (128) op)
    constant bits(4*8) in0 = op< 96+:8> : op< 64+:8> : op< 32+:8> : op<  0+:8>;
    constant bits(4*8) in1 = op<104+:8> : op< 72+:8> : op< 40+:8> : op<  8+:8>;
    constant bits(4*8) in2 = op<112+:8> : op< 80+:8> : op< 48+:8> : op< 16+:8>;
    constant bits(4*8) in3 = op<120+:8> : op< 88+:8> : op< 56+:8> : op< 24+:8>;

    bits(4*8) out0;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    bits(4*8) out1;
    bits(4*8) out2;
    bits(4*8) out3;

    for c = 0 to 3
        out0<c*8+:8> = (FFmul02(in0<c*8+:8>) EOR FFmul03(in1<c*8+:8>) EOR
                                in2<c*8+:8>  EOR         in3<c*8+:8>);
        out1<c*8+:8> = (FFmul02(in1<c*8+:8>)  EOR FFmul03(in2<c*8+:8>) EOR
                                in3<c*8+:8>   EOR         in0<c*8+:8>);
        out2<c*8+:8> = (FFmul02(in2<c*8+:8>)  EOR FFmul03(in3<c*8+:8>) EOR
                                in0<c*8+:8>   EOR         in1<c*8+:8>);
        out3<c*8+:8> = (FFmul02(in3<c*8+:8>)  EOR FFmul03(in0<c*8+:8>) EOR
                                in1<c*8+:8>   EOR         in2<c*8+:8>);

    return (
        out3<3*8+:8> : out2<3*8+:8> : out1<3*8+:8> : out0<3*8+:8> :
        out3<2*8+:8> : out2<2*8+:8> : out1<2*8+:8> : out0<2*8+:8> :
        out3<1*8+:8> : out2<1*8+:8> : out1<1*8+:8> : out0<1*8+:8> :
        out3<0*8+:8> : out2<0*8+:8> : out1<0*8+:8> : out0<0*8+:8>
    );
```

G1.2.3.81 AESShiftRows

```
// AESShiftRows()
// ==============
// Transformation in the Cipher that processes the State by cyclically
// shifting the last three rows of the State by different offsets.

bits(128) AESShiftRows(bits(128) op)
    return (
        op< 95: 88> : op< 55: 48> : op< 15:  8> : op<103: 96> :
        op< 63: 56> : op< 23: 16> : op<111:104> : op< 71: 64> :
        op< 31: 24> : op<119:112> : op< 79: 72> : op< 39: 32> :
        op<127:120> : op< 87: 80> : op< 47: 40> : op<  7:  0>
    );
```

G1.2.3.82 AESSubBytes

```
// AESSubBytes()
// =============
// Transformation in the Cipher that processes the State using a nonlinear
// byte substitution table (S-box) that operates on each of the State bytes
// independently.

bits(128) AESSubBytes(bits(128) op)
    // S-box values
    constant bits(16*16*8) GF2 = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0x16bb54b00f2d99416842e6bf0d89a18c<127:0> :
        /*E*/ 0xdf2855cee9871e9b948ed9691198f8e1<127:0> :
        /*D*/ 0x9e1dc186b95735610ef6034866b53e70<127:0> :
        /*C*/ 0x8a8bbd4b1f74dde8c6b4a61c2e2578ba<127:0> :
        /*B*/ 0x08ae7a65eaf4566ca94ed58d6d37c8e7<127:0> :
        /*A*/ 0x79e4959162acd3c25c2406490a3a32e0<127:0> :
        /*9*/ 0xdb0b5ede14b8ee4688902a22dc4f8160<127:0> :
        /*8*/ 0x73195d643d7ea7c41744975fec130ccd<127:0> :
        /*7*/ 0xd2f3ff1021dab6bcf5389d928f40a351<127:0> :
        /*6*/ 0xa89f3c507f02f94585334d43fbaaefd0<127:0> :
        /*5*/ 0xcf584c4a39becb6a5bb1fc20ed00d153<127:0> :
        /*4*/ 0x842fe329b3d63b52a05a6e1b1a2c8309<127:0> :
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        /*3*/ 0x75b227ebe28012079a059618c323c704<127:0> :
        /*2*/ 0x1531d871f1e5a534ccf73f362693fdb7<127:0> :
        /*1*/ 0xc072a49cafa2d4adf04759fa7dc982ca<127:0> :
        /*0*/ 0x76abd7fe2b670130c56f6bf27b777c63<127:0>
    );
    bits(128) out;
    for i = 0 to 15
        out<i*8+:8> = GF2<UInt(op<i*8+:8>)*8+:8>;
    return out;
```

G1.2.3.83 FFmul02

```
// FFmul02()
// =========

bits(8) FFmul02(bits(8) b)
    constant bits(256*8) FFmul_02 = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0xE5E7E1E3EDEFE9EBF5F7F1F3FDFFF9FB<127:0> :
        /*E*/ 0xC5C7C1C3CDCFC9CBD5D7D1D3DDDFD9DB<127:0> :
        /*D*/ 0xA5A7A1A3ADAFA9ABB5B7B1B3BDBFB9BB<127:0> :
        /*C*/ 0x858781838D8F898B959791939D9F999B<127:0> :
        /*B*/ 0x656761636D6F696B757771737D7F797B<127:0> :
        /*A*/ 0x454741434D4F494B555751535D5F595B<127:0> :
        /*9*/ 0x252721232D2F292B353731333D3F393B<127:0> :
        /*8*/ 0x050701030D0F090B151711131D1F191B<127:0> :
        /*7*/ 0xFEFCFAF8F6F4F2F0EEECEAE8E6E4E2E0<127:0> :
        /*6*/ 0xDEDCDAD8D6D4D2D0CECCCAC8C6C4C2C0<127:0> :
        /*5*/ 0xBEBCBAB8B6B4B2B0AEACAAA8A6A4A2A0<127:0> :
        /*4*/ 0x9E9C9A98969492908E8C8A8886848280<127:0> :
        /*3*/ 0x7E7C7A78767472706E6C6A6866646260<127:0> :
        /*2*/ 0x5E5C5A58565452504E4C4A4846444240<127:0> :
        /*1*/ 0x3E3C3A38363432302E2C2A2826242220<127:0> :
        /*0*/ 0x1E1C1A18161412100E0C0A0806040200<127:0>
    );
    return FFmul_02<UInt(b)*8+:8>;
```

G1.2.3.84 FFmul03

```
// FFmul03()
// =========

bits(8) FFmul03(bits(8) b)
    constant bits(256*8) FFmul_03 = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0x1A191C1F16151013020104070E0D080B<127:0> :
        /*E*/ 0x2A292C2F26252023323134373E3D383B<127:0> :
        /*D*/ 0x7A797C7F76757073626164676E6D686B<127:0> :
        /*C*/ 0x4A494C4F46454043525154575E5D585B<127:0> :
        /*B*/ 0xDAD9DCDFD6D5D0D3C2C1C4C7CECDC8CB<127:0> :
        /*A*/ 0xEAE9ECEFE6E5E0E3F2F1F4F7FEFDF8FB<127:0> :
        /*9*/ 0xBAB9BCBFB6B5B0B3A2A1A4A7AEADA8AB<127:0> :
        /*8*/ 0x8A898C8F868580839291949 79E9D989B<127:0> :
        /*7*/ 0x818287848D8E8B88999A9F9C95969390<127:0> :
        /*6*/ 0xB1B2B7B4BDBEBBB8A9AAAFACA5A6A3A0<127:0> :
        /*5*/ 0xE1E2E7E4EDEEEBE8F9FAFFFCF5F6F3F0<127:0> :
        /*4*/ 0xD1D2D7D4DDDEDBD8C9CACFCCC5C6C3C0<127:0> :
        /*3*/ 0x414247444D4E4B48595A5F5C55565350<127:0> :
        /*2*/ 0x717277747D7E7B78696A6F6C65666360<127:0> :
        /*1*/ 0x212227242D2E2B28393A3F3C35363330<127:0> :
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        /*0*/ 0x111217141D1E1B18090A0F0C05060300<127:0>
    );
    return FFmul_03<UInt(b)*8+:8>;
```

G1.2.3.85 FFmul09

```
// FFmul09()
// =========

bits(8) FFmul09(bits(8) b)
    constant bits(256*8) FFmul_09 = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0x464F545D626B70790E071C152A233831<127:0> :
        /*E*/ 0xD6DFC4CDF2FBE0E99E978C85BAB3A8A1<127:0> :
        /*D*/ 0x7D746F6659504B42353C272E1118030A<127:0> :
        /*C*/ 0xEDE4FFF6C9C0DBD2A5ACB7BE8188939A<127:0> :
        /*B*/ 0x3039222B141D060F78716A635C554E47<127:0> :
        /*A*/ 0xA0A9B2BB848D969FE8E1FAF3CCC5DED7<127:0> :
        /*9*/ 0x0B0219102F263D34434A5158676E757C<127:0> :
        /*8*/ 0x9B928980BFB6ADA4D3DAC1C8F7FEE5EC<127:0> :
        /*7*/ 0xAAA3B8B18E879C95E2EBF0F9C6CFD4DD<127:0> :
        /*6*/ 0x3A3328211E170C05727B6069565F444D<127:0> :
        /*5*/ 0x9198838AB5BCA7AED9D0CBC2FDF4EFE6<127:0> :
        /*4*/ 0x0108131A252C373E49405B526D647F76<127:0> :
        /*3*/ 0xDCD5CEC7F8F1EAE3949D868FB0B9A2AB<127:0> :
        /*2*/ 0x4C455E5768617A73040D161F2029323B<127:0> :
        /*1*/ 0xE7EEF5FCC3CAD1D8AFA6BDB48B829990<127:0> :
        /*0*/ 0x777E656C535A41483F362D241B120900<127:0>
    );
    return FFmul_09<UInt(b)*8+:8>;
```

G1.2.3.86 FFmul0B

```
// FFmul0B()
// =========

bits(8) FFmul0B(bits(8) b)
    constant bits(256*8) FFmul_0B = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0xA3A8B5BE8F849992FBF0EDE6D7DCC1CA<127:0> :
        /*E*/ 0x1318050E3F3429224B405D56676C717A<127:0> :
        /*D*/ 0xD8D3CEC5F4FFE2E9808B969DACA7BAB1<127:0> :
        /*C*/ 0x68637E75444F5259303B262D1C170A01<127:0> :
        /*B*/ 0x555E434879726F640D061B10212A373C<127:0> :
        /*A*/ 0xE5EEF3F8C9C2DFD4BDB6ABA0919A878C<127:0> :
        /*9*/ 0x2E2538330209141F767D606B5A514C47<127:0> :
        /*8*/ 0x9E958883B2B9A4AFC6CDD0DBEAE1FCF7<127:0> :
        /*7*/ 0x545F424978736E650C071A11202B363D<127:0> :
        /*6*/ 0xE4EFF2F9C8C3DED5BCB7AAA1909B868D<127:0> :
        /*5*/ 0x2F2439320308151E777C616A5B504D46<127:0> :
        /*4*/ 0x9F948982B3B8A5AEC7CCD1DAEBE0FDF6<127:0> :
        /*3*/ 0xA2A9B4BF8E859893FAF1ECE7D6DDC0CB<127:0> :
        /*2*/ 0x1219040F3E3528234A415C57666D707B<127:0> :
        /*1*/ 0xD9D2CFC4F5FEE3E8818A979CADA6BBB0<127:0> :
        /*0*/ 0x69627F74454E5358313A272C1D160B00<127:0>
    );
    return FFmul_0B<UInt(b)*8+:8>;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.3.87 FFmul0D

```
// FFmul0D()
// =========

bits(8) FFmul0D(bits(8) b)
    constant bits(256*8) FFmul_0D = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0x979A8D80A3AEB9B4FFF2E5E8CBC6D1DC<127:0> :
        /*E*/ 0x474A5D50737E69642F2235381B16010C<127:0> :
        /*D*/ 0x2C21363B1815020F44495E53707D6A67<127:0> :
        /*C*/ 0xFCF1E6EBC8C5D2DF94998E83A0ADBAB7<127:0> :
        /*B*/ 0xFAF7E0EDCEC3D4D9929F8885A6ABBCB1<127:0> :
        /*A*/ 0x2A27303D1E130409424F5855767B6C61<127:0> :
        /*9*/ 0x414C5B5675786F622924333E1D10070A<127:0> :
        /*8*/ 0x919C8B86A5A8BFB2F9F4E3EECDC0D7DA<127:0> :
        /*7*/ 0x4D40575A7974636E25283F32111C0B06<127:0> :
        /*6*/ 0x9D90878AA9A4B3BEF5F8EFE2C1CCDBD6<127:0> :
        /*5*/ 0xF6FBECE1C2CFD8D59E938489AAA7B0BD<127:0> :
        /*4*/ 0x262B3C31121F08054E4354597A77606D<127:0> :
        /*3*/ 0x202D3A3714190E034845525F7C71666B<127:0> :
        /*2*/ 0xF0FDEAE7C4C9DED39895828FACA1B6BB<127:0> :
        /*1*/ 0x9B96818CAFA2B5B8F3FEE9E4C7CADDD0<127:0> :
        /*0*/ 0x4B46515C7F726568232E3934171A0D00<127:0>
    );
    return FFmul_0D<UInt(b)*8+:8>;
```

G1.2.3.88 FFmul0E

```
// FFmul0E()
// =========

bits(8) FFmul0E(bits(8) b)
    constant bits(256*8) FFmul_0E = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0x8D83919FB5BBA9A7FDF3E1EFC5CBD9D7<127:0> :
        /*E*/ 0x6D63717F555B49471D13010F252B3937<127:0> :
        /*D*/ 0x56584A446E60727C26283A341E10020C<127:0> :
        /*C*/ 0xB6B8AAA48E80929CC6C8DAD4FEF0E2EC<127:0> :
        /*B*/ 0x202E3C321816040A505E4C426866747A<127:0> :
        /*A*/ 0xC0CEDCD2F8F6E4EAB0BEACA28886949A<127:0> :
        /*9*/ 0xFBF5E7E9C3CDDFD18B859799B3BDAFA1<127:0> :
        /*8*/ 0x1B150709232D3F316B657779535D4F41<127:0> :
        /*7*/ 0xCCC2D0DEF4FAE8E6BCB2A0AE848A9896<127:0> :
        /*6*/ 0x2C22303E141A08065C52404E646A7876<127:0> :
        /*5*/ 0x17190B052F21333D67697B755F51434D<127:0> :
        /*4*/ 0xF7F9EBE5CFC1D3DD87899B95BFB1A3AD<127:0> :
        /*3*/ 0x616F7D735957454B111F0D032927353B<127:0> :
        /*2*/ 0x818F9D93B9B7A5ABF1FFEDE3C9C7D5DB<127:0> :
        /*1*/ 0xBAB4A6A8828C9E90CAC4D6D8F2FCEEE0<127:0> :
        /*0*/ 0x5A544648626C7E702A243638121C0E00<127:0>
    );
    return FFmul_0E<UInt(b)*8+:8>;
```

G1.2.3.89 SHA256hash

```
// SHA256hash()
// ============

bits(128) SHA256hash(bits (128) x_in, bits(128) y_in, bits(128) w, boolean part1)
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    bits(32) chs, maj, t;
    bits(128) x = x_in;
    bits(128) y = y_in;

    for e = 0 to 3
        chs = SHAchoose(y<31:0>, y<63:32>, y<95:64>);
        maj = SHAmajority(x<31:0>, x<63:32>, x<95:64>);
        t = y<127:96> + SHAhashSIGMA1(y<31:0>) + chs + Elem[w, e, 32];
        x<127:96> = t + x<127:96>;
        y<127:96> = t + SHAhashSIGMA0(x<31:0>) + maj;
        constant bits(256) yx = ROL(y : x, 32);
        (y, x) = (yx<128+:128>, yx<0+:128>);
    return (if part1 then x else y);
```

G1.2.3.90 SHAchoose

```
// SHAchoose()
// ===========

bits(32) SHAchoose(bits(32) x, bits(32) y, bits(32) z)
    return (((y EOR z) AND x) EOR z);
```

G1.2.3.91 SHAhashSIGMA0

```
// SHAhashSIGMA0()
// ===============

bits(32) SHAhashSIGMA0(bits(32) x)
    return ROR(x, 2) EOR ROR(x, 13) EOR ROR(x, 22);
```

G1.2.3.92 SHAhashSIGMA1

```
// SHAhashSIGMA1()
// ===============

bits(32) SHAhashSIGMA1(bits(32) x)
    return ROR(x, 6) EOR ROR(x, 11) EOR ROR(x, 25);
```

G1.2.3.93 SHAmajority

```
// SHAmajority()
// =============

bits(N) SHAmajority(bits(N) x, bits(N) y, bits(N) z)
    assert N IN {32, 64};
    return ((x AND y) OR ((x OR y) AND z));
```

G1.2.3.94 SHAparity

```
// SHAparity()
// ===========
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
bits(32) SHAparity(bits(32) x, bits(32) y, bits(32) z)
    return (x EOR y EOR z);
```

G1.2.3.95 Sbox

```
// Sbox()
// ======
// Used in SM4E crypto instruction

bits(8) Sbox(bits(8) sboxin)
    bits(8) sboxout;
    constant bits(2048) sboxstring = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0xd690e9fecce13db716b614c228fb2c05<127:0> :
        /*E*/ 0x2b679a762abe04c3aa44132649860699<127:0> :
        /*D*/ 0x9c4250f491ef987a33540b43edcfac62<127:0> :
        /*C*/ 0xe4b31ca9c908e89580df94fa758f3fa6<127:0> :
        /*B*/ 0x4707a7fcf37317ba83593c19e6854fa8<127:0> :
        /*A*/ 0x686b81b27164da8bf8eb0f4b70569d35<127:0> :
        /*9*/ 0x1e240e5e6358d1a225227c3b01217887<127:0> :
        /*8*/ 0xd40046579fd327524c3602e7a0c4c89e<127:0> :
        /*7*/ 0xeabf8ad240c738b5a3f7f2cef96115a1<127:0> :
        /*6*/ 0xe0ae5da49b341a55ad933230f58cb1e3<127:0> :
        /*5*/ 0x1df6e22e8266ca60c02923ab0d534e6f<127:0> :
        /*4*/ 0xd5db3745defd8e2f03ff6a726d6c5b51<127:0> :
        /*3*/ 0x8d1baf92bbddbc7f11d95c411f105ad8<127:0> :
        /*2*/ 0x0ac13188a5cd7bbd2d74d012b8e5b4b0<127:0> :
        /*1*/ 0x8969974a0c96777e65b9f109c56ec684<127:0> :
        /*0*/ 0x18f07dec3adc4d2079ee5f3ed7cb3948<127:0>
    );
    constant integer sboxindex = 255 - UInt(sboxin);
    sboxout = Elem[sboxstring, sboxindex, 8];
    return sboxout;
```

G1.2.3.96 DecodeType

```
// DecodeType
// ==========

enumeration DecodeType {
    Decode_UNDEF,
    Decode_NOP,
    Decode_OK
};
```

G1.2.3.97 EndOfDecode

```
// EndOfDecode()
// =============
// This function is invoked to end the Decode phase and performs Branch target Checks
// before taking any UNDEFINED exceptions, NOPs, or continuing to execute.

EndOfDecode(DecodeType reason)
    case reason of
        when Decode_NOP   ExecuteAsNOP();
        when Decode_UNDEF Undefined();
        when Decode_OK                  // Continue to execute.
    return;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.3.98 ClearExclusiveByAddress

```
// ClearExclusiveByAddress()
// =========================
// Clear the global Exclusives monitors for all PEs EXCEPT processorid if they
// record any part of the physical address region of size bytes starting at paddress.
// It is IMPLEMENTATION DEFINED whether the global Exclusives monitor for processorid
// is also cleared if it records any part of the address region.

ClearExclusiveByAddress(FullAddress paddress, integer processorid, integer size);
```

G1.2.3.99 ClearExclusiveLocal

```
// ClearExclusiveLocal()
// =====================
// Clear the local Exclusives monitor for the specified processorid.

ClearExclusiveLocal(integer processorid);
```

G1.2.3.100 ExclusiveMonitorsStatus

```
// ExclusiveMonitorsStatus()
// =========================
// Returns '0' to indicate success if the last memory write by this PE was to
// the same physical address region endorsed by ExclusiveMonitorsPass().
// Returns '1' to indicate failure if address translation resulted in a different
// physical address.

bit ExclusiveMonitorsStatus();
```

G1.2.3.101 IsExclusiveGlobal

```
// IsExclusiveGlobal()
// ===================
// Return TRUE if the global Exclusives monitor for processorid includes all of
// the physical address region of size bytes starting at paddress.

boolean IsExclusiveGlobal(FullAddress paddress, integer processorid, integer size);
```

G1.2.3.102 IsExclusiveLocal

```
// IsExclusiveLocal()
// ==================
// Return TRUE if the local Exclusives monitor for processorid includes all of
// the physical address region of size bytes starting at paddress.

boolean IsExclusiveLocal(FullAddress paddress, integer processorid, integer size);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.3.103 MarkExclusiveGlobal

```
// MarkExclusiveGlobal()
// =====================
// Record the physical address region of size bytes starting at paddress in
// the global Exclusives monitor for processorid.

MarkExclusiveGlobal(FullAddress paddress, integer processorid, integer size);
```

G1.2.3.104 MarkExclusiveLocal

```
// MarkExclusiveLocal()
// ====================
// Record the physical address region of size bytes starting at paddress in
// the local Exclusives monitor for processorid.

MarkExclusiveLocal(FullAddress paddress, integer processorid, integer size);
```

G1.2.3.105 ProcessorID

```
// ProcessorID()
// =============
// Return the ID of the currently executing PE.

integer ProcessorID();
```

G1.2.3.106 ArchHasVMSAExtension

```
// ArchHasVMSAExtension()
// ======================

boolean  ArchHasVMSAExtension()
    return HaveEL1VMSAExt();
```

G1.2.3.107 HaveEL1VMSAExt

```
// HaveEL1VMSAExt()
// ================
// Returns TRUE if VMSA is supported at stage1 EL1&0 translation regime, FALSE otherwise.

boolean HaveEL1VMSAExt()
    return ID_AA64MMFR0_EL1.MSA == '1111' && ID_AA64MMFR0_EL1.MSA_frac == '0010';
```

G1.2.3.108 HaveSoftwareLock

```
// HaveSoftwareLock()
// ==================
// Returns TRUE if Software Lock is implemented.

boolean HaveSoftwareLock(Component component)
    if IsFeatureImplemented(FEAT_Debugv8p4) then
        return FALSE;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if IsFeatureImplemented(FEAT_DoPD) && component != Component_CTI then
        return FALSE;
    case component of
        when Component_Debug
            return boolean IMPLEMENTATION_DEFINED "Debug has Software Lock";
        when Component_PMU
            return boolean IMPLEMENTATION_DEFINED "PMU has Software Lock";
        when Component_CTI
            return boolean IMPLEMENTATION_DEFINED "CTI has Software Lock";
        otherwise
            Unreachable();
```

G1.2.3.109 HaveTraceExt

```
// HaveTraceExt()
// ==============
// Returns TRUE if Trace functionality as described by the Trace Architecture
// is implemented.

boolean HaveTraceExt()
    return IsFeatureImplemented(FEAT_ETMv4);
```

G1.2.3.110 InsertIESBBeforeException

```
// InsertIESBBeforeException()
// ===========================
// Returns an implementation defined choice whether to insert an implicit error synchronization
// barrier before exception.
// If SCTLR_ELx.IESB is 1 when an exception is generated to ELx, any pending Unrecoverable
// SError interrupt must be taken before executing any instructions in the exception handler.
// However, this can be before the branch to the exception handler is made.

boolean InsertIESBBeforeException(bits(2) el)
    return (IsFeatureImplemented(FEAT_IESB) && boolean IMPLEMENTATION_DEFINED
            "Has Implicit Error Synchronization Barrier before Exception");
```

G1.2.3.111 ActionRequired

```
// ActionRequired()
// ================
// Return an implementation specific value:
// returns TRUE if action is required, FALSE otherwise.

boolean ActionRequired();
```

G1.2.3.112 ClearPendingPhysicalSError

```
// ClearPendingPhysicalSError()
// ============================
// Clear a pending physical SError interrupt.

ClearPendingPhysicalSError();
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.3.113 ClearPendingVirtualSError

```
// ClearPendingVirtualSError()
// ===========================
// Clear a pending virtual SError interrupt.

ClearPendingVirtualSError()
    HCR_EL2.VSE = '0';
```

G1.2.3.114 ErrorIsContained

```
// ErrorIsContained()
// ==================
// Return an implementation specific value:
// TRUE if Error is contained by the PE, FALSE otherwise.

boolean ErrorIsContained();
```

G1.2.3.115 ErrorIsSynchronized

```
// ErrorIsSynchronized()
// =====================
// Return an implementation specific value:
// returns TRUE if Error is synchronized by any synchronization event
// FALSE otherwise.

boolean ErrorIsSynchronized();
```

G1.2.3.116 ExtAbortToAArch64

```
// ExtAbortToAArch64()
// ===================
// Returns TRUE if synchronous exception is being taken to an Exception level using AArch64.

boolean ExtAbortToAArch64(FaultRecord fault)
    assert IsExternalSyncAbort(fault.statuscode);

    return !ELUsingAArch32(SyncExternalAbortTarget(fault));
```

G1.2.3.117 ExternalAbort

```
// ExternalAbort()
// ===============

ExternalAbort(FaultRecord fault)
    if IsExternalSyncAbort(fault) then
        AArch64.Abort(fault);
    else
        PendSErrorInterrupt(fault);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.3.118 ExternalFault

```
// ExternalFault()
// ===============
// Return a fault recording indicating a fault for a Synchronous/Asynchronous External abort.

FaultRecord ExternalFault(PhysMemRetStatus memretstatus, boolean iswrite,
                          AddressDescriptor memaddrdesc, integer size, AccessDescriptor accdesc)
    assert (memretstatus.statuscode IN {Fault_SyncExternal, Fault_AsyncExternal} ||
           (!IsFeatureImplemented(FEAT_RAS) && memretstatus.statuscode IN {Fault_SyncParity,
                                                                           Fault_AsyncParity}));

    fault            = NoFault(accdesc, memaddrdesc.vaddress);
    fault.statuscode = memretstatus.statuscode;
    fault.write      = iswrite;
    fault.extflag    = memretstatus.extflag;
    // It is implementation specific whether External aborts signaled
    // in-band synchronously are taken synchronously or asynchronously
    if (IsExternalSyncAbort(fault) &&
            !IsExternalAbortTakenSynchronously(memretstatus, iswrite, memaddrdesc,
                                               size, accdesc)) then
        if fault.statuscode == Fault_SyncParity then
            fault.statuscode = Fault_AsyncParity;
        else
            fault.statuscode = Fault_AsyncExternal;

    if IsFeatureImplemented(FEAT_RAS) then
        fault.merrorstate = memretstatus.merrorstate;

    return fault;
```

G1.2.3.119 FaultIsCorrected

```
// FaultIsCorrected()
// ==================
// Return an implementation specific value:
// TRUE if fault is corrected by the PE, FALSE otherwise.

boolean FaultIsCorrected();
```

G1.2.3.120 GetPendingPhysicalSError

```
// GetPendingPhysicalSError()
// ==========================
// Returns the FaultRecord containing details of pending Physical SError
// interrupt.

FaultRecord GetPendingPhysicalSError();
```

G1.2.3.121 HandleExternalAbort

```
// HandleExternalAbort()
// =====================
// Takes a Synchronous/Asynchronous abort based on fault.

HandleExternalAbort(PhysMemRetStatus memretstatus, boolean iswrite,
                    AddressDescriptor memaddrdesc, integer size,
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                    AccessDescriptor accdesc)
    assert (memretstatus.statuscode IN {Fault_SyncExternal, Fault_AsyncExternal} ||
           (!IsFeatureImplemented(FEAT_RAS) && memretstatus.statuscode IN {Fault_SyncParity,
                                                                           Fault_AsyncParity}));

    constant FaultRecord fault = ExternalFault(memretstatus, iswrite, memaddrdesc, size, accdesc);
    ExternalAbort(fault);
```

G1.2.3.122 HandleExternalReadAbort

```
// HandleExternalReadAbort()
// =========================
// Wrapper function for HandleExternalAbort function in case of an External
// Abort on memory read.

HandleExternalReadAbort(PhysMemRetStatus memstatus, AddressDescriptor memaddrdesc,
                        integer size, AccessDescriptor accdesc)
    iswrite = FALSE;
    HandleExternalAbort(memstatus, iswrite, memaddrdesc, size, accdesc);
```

G1.2.3.123 HandleExternalTTWAbort

```
// HandleExternalTTWAbort()
// ========================
// Take Asynchronous abort or update FaultRecord for Translation Table Walk
// based on PhysMemRetStatus.

FaultRecord HandleExternalTTWAbort(PhysMemRetStatus memretstatus, boolean iswrite,
                                   AddressDescriptor memaddrdesc,
                                   AccessDescriptor accdesc, integer size,
                                   FaultRecord input_fault)
    output_fault = input_fault;
    output_fault.extflag = memretstatus.extflag;
    output_fault.statuscode = memretstatus.statuscode;
    if (IsExternalSyncAbort(output_fault) &&
            !IsExternalAbortTakenSynchronously(memretstatus, iswrite, memaddrdesc,
                                               size, accdesc)) then
        if output_fault.statuscode == Fault_SyncParity then
            output_fault.statuscode = Fault_AsyncParity;
        else
            output_fault.statuscode = Fault_AsyncExternal;

    // If a synchronous fault is on a translation table walk, then update the fault type.
    if IsExternalSyncAbort(output_fault) then
        if output_fault.statuscode == Fault_SyncParity then
            output_fault.statuscode = Fault_SyncParityOnWalk;
        else
            output_fault.statuscode = Fault_SyncExternalOnWalk;
    if IsFeatureImplemented(FEAT_RAS) then
        output_fault.merrorstate = memretstatus.merrorstate;
    if !IsExternalSyncAbort(output_fault) then
        PendSErrorInterrupt(output_fault);
        output_fault.statuscode = Fault_None;
    return output_fault;
```

G1.2.3.124 HandleExternalWriteAbort

```
// HandleExternalWriteAbort()
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// ==========================
// Wrapper function for HandleExternalAbort function in case of an External
// Abort on memory write.

HandleExternalWriteAbort(PhysMemRetStatus memstatus, AddressDescriptor memaddrdesc,
                         integer size, AccessDescriptor accdesc)
    iswrite = TRUE;
    HandleExternalAbort(memstatus, iswrite, memaddrdesc, size, accdesc);
```

G1.2.3.125 IsExternalAbortTakenSynchronously

```
// IsExternalAbortTakenSynchronously()
// ===================================
// Return an implementation specific value:
// TRUE if the fault returned for the access can be taken synchronously,
// FALSE otherwise.
//
// This might vary between accesses, for example depending on the error type
// or memory type being accessed.
// External aborts on data accesses and translation table walks on data accesses
// can be either synchronous or asynchronous.
//
// When FEAT_DoubleFault is not implemented, External aborts on instruction
// fetches and translation table walks on instruction fetches can be either
// synchronous or asynchronous.
// When FEAT_DoubleFault is implemented, all External abort exceptions on
// instruction fetches and translation table walks on instruction fetches
// must be synchronous.

boolean IsExternalAbortTakenSynchronously(PhysMemRetStatus memstatus, boolean iswrite,
                                          AddressDescriptor desc, integer size,
                                          AccessDescriptor accdesc);
```

G1.2.3.126 IsPhysicalSErrorPending

```
// IsPhysicalSErrorPending()
// =========================
// Returns TRUE if a physical SError interrupt is pending.

boolean IsPhysicalSErrorPending();
```

G1.2.3.127 IsSErrorEdgeTriggered

```
// IsSErrorEdgeTriggered()
// =======================
// Returns TRUE if the physical SError interrupt is edge-triggered
// and FALSE otherwise.

boolean IsSErrorEdgeTriggered()
    return boolean IMPLEMENTATION_DEFINED "Edge-triggered SError";
```

G1.2.3.128 IsSynchronizablePhysicalSErrorPending

```
// IsSynchronizablePhysicalSErrorPending()
// =======================================
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// Returns TRUE if a synchronizable physical SError interrupt is pending.

boolean IsSynchronizablePhysicalSErrorPending();
```

G1.2.3.129 IsVirtualSErrorPending

```
// IsVirtualSErrorPending()
// ========================
// Return TRUE if a virtual SError interrupt is pending.

boolean IsVirtualSErrorPending()
    return HCR_EL2.VSE == '1';
```

G1.2.3.130 PEErrorState

```
// PEErrorState()
// ==============
// Returns the error state of the PE on taking an error exception:
// The PE error state reported to software through the exception syndrome also
// depends on how the exception is taken, and so might differ from the value
// returned from this function.

ErrorState PEErrorState(FaultRecord fault)
    assert !FaultIsCorrected();
    if (!ErrorIsContained() ||
        (!ErrorIsSynchronized() && !StateIsRecoverable()) ||
         ReportErrorAsUC()) then
        return ErrorState_UC;

    if !StateIsRecoverable() || ReportErrorAsUEU() then
        return ErrorState_UEU;

    if ActionRequired() || ReportErrorAsUER() then
        return ErrorState_UER;

    return ErrorState_UEO;
```

G1.2.3.131 PendSErrorInterrupt

```
// PendSErrorInterrupt()
// =====================
// Pend the SError Interrupt.

PendSErrorInterrupt(FaultRecord fault);
```

G1.2.3.132 ReportErrorAsIMPDEF

```
// ReportErrorAsIMPDEF()
// =====================
// Return an implementation specific value:
// returns TRUE if Error is IMPDEF, FALSE otherwise.

boolean ReportErrorAsIMPDEF();
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.3.133 ReportErrorAsUC

```
// ReportErrorAsUC()
// =================
// Return an implementation specific value:
// returns TRUE if Error is Uncontainable, FALSE otherwise.

boolean ReportErrorAsUC();
```

G1.2.3.134 ReportErrorAsUER

```
// ReportErrorAsUER()
// ==================
// Return an implementation specific value:
// returns TRUE if Error is Recoverable, FALSE otherwise.

boolean ReportErrorAsUER();
```

G1.2.3.135 ReportErrorAsUEU

```
// ReportErrorAsUEU()
// ==================
// Return an implementation specific value:
// returns TRUE if Error is Unrecoverable, FALSE otherwise.

boolean ReportErrorAsUEU();
```

G1.2.3.136 ReportErrorAsUncategorized

```
// ReportErrorAsUncategorized()
// ============================
// Return an implementation specific value:
// returns TRUE if Error is uncategorized, FALSE otherwise.

boolean ReportErrorAsUncategorized();
```

G1.2.3.137 StateIsRecoverable

```
// StateIsRecoverable()
// ====================
// Return an implementation specific value:
// returns TRUE is PE State is unrecoverable else FALSE.

boolean StateIsRecoverable();
```

G1.2.3.138 FixedToFP

```
// FixedToFP()
// ===========

// Convert M-bit fixed point 'op' with FBITS fractional bits to
// N-bit precision floating point, controlled by UNSIGNED and ROUNDING.
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
bits(N) FixedToFP(bits(M) op, integer fbits, boolean unsigned, FPCR_Type fpcr,
                  FPRounding rounding, integer N)
    assert N IN {16,32,64};
    assert M IN {16,32,64};
    bits(N) result;
    assert fbits >= 0;
    assert rounding != FPRounding_ODD;

    // Correct signed-ness
    int_operand = if unsigned then UInt(op) else SInt(op);

    // Scale by fractional bits and generate a real value
    real_operand = Real(int_operand) / 2.0^fbits;

    if real_operand == 0.0 then
        result = FPZero('0', N);
    else
        result = FPRound(real_operand, fpcr, rounding, N);

    return result;
```

G1.2.3.139 FPAbs

```
// FPAbs()
// =======

bits(N) FPAbs(bits(N) op, FPCR_Type fpcr)
    assert N IN {16,32,64};
    return '0' : op<N-2:0>;
```

G1.2.3.140 FPAdd

```
// FPAdd()
// =======

bits(N) FPAdd(bits(N) op1, bits(N) op2, FPCR_Type fpcr)

    assert N IN {16,32,64};
    rounding = FPRoundingMode(fpcr);

    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1  = (type1 == FPType_Infinity);  inf2  = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);      zero2 = (type2 == FPType_Zero);
        if inf1 && inf2 && sign1 == NOT(sign2) then
            result = FPDefaultNaN(fpcr, N);
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
            result = FPInfinity('0', N);
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
            result = FPInfinity('1', N);
        elsif zero1 && zero2 && sign1 == sign2 then
            result = FPZero(sign1, N);
        else
            result_value = value1 + value2;
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign, N);
            else
                result = FPRound(result_value, fpcr, rounding, N);

    return result;
```

G1.2.3.141 FPBits

```
// FPBits()
// ========
// Returns the minimum exponent, numbers of exponent and fraction bits.

FPBitsType FPBits(integer N)
    FPFracBits F;
    integer minimum_exp;
    if N == 16 then
        minimum_exp = -14;   F = 10;
    elsif N == 32 then
        minimum_exp = -126;  F = 23;
    else  // N == 64
        minimum_exp = -1022; F = 52;

    return (F, minimum_exp);
```

G1.2.3.142 FPBitsType

```
// FPBitsType
// ==========

type FPBitsType = (FPFracBits, integer);
```

G1.2.3.143 FPFracBits

```
// FPFracBits
// ==========

type FPFracBits = integer;
```

G1.2.3.144 FPCompare

```
// FPCompare()
// ===========

bits(4) FPCompare(bits(N) op1, bits(N) op2, boolean signal_nans, FPCR_Type fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    bits(4) result;
    if type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN} then
        result = '0011';
        if type1 == FPType_SNaN || type2 == FPType_SNaN || signal_nans then
            FPProcessException(FPExc_InvalidOp, fpcr);
    else
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        if value1 == value2 then
            result = '0110';
        elsif value1 < value2 then
            result = '1000';
        else  // value1 > value2
            result = '0010';

    return result;
```

G1.2.3.145 FPCompareEQ

```
// FPCompareEQ()
// =============

boolean FPCompareEQ(bits(N) op1, bits(N) op2, FPCR_Type fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    boolean result;
    if type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN} then
        result = FALSE;
        if type1 == FPType_SNaN || type2 == FPType_SNaN then
            FPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 == value2);
    return result;
```

G1.2.3.146 FPCompareGE

```
// FPCompareGE()
// =============

boolean FPCompareGE(bits(N) op1, bits(N) op2, FPCR_Type fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    boolean result;
    if type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN} then
        result = FALSE;
        FPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 >= value2);
    return result;
```

G1.2.3.147 FPCompareGT

```
// FPCompareGT()
// =============

boolean FPCompareGT(bits(N) op1, bits(N) op2, FPCR_Type fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    boolean result;
    if type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN} then
        result = FALSE;
        FPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 > value2);
    return result;
```

G1.2.3.148 FPConvert

```
// FPConvert()
// ===========

// Convert floating point 'op' with N-bit precision to M-bit precision,
// with rounding controlled by ROUNDING.
// This is used by the FP-to-FP conversion instructions and so for
// half-precision data ignores FZ16, but observes AHP.

bits(M) FPConvert(bits(N) op, FPCR_Type fpcr, FPRounding rounding, integer M)

    assert M IN {16,32,64};
    assert N IN {16,32,64};
    bits(M) result;

    // Unpack floating-point operand optionally with flush-to-zero.
    (fptype,sign,value) = FPUnpackCV(op, fpcr);

    alt_hp = (M == 16) && (fpcr.AHP == '1');

    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        if alt_hp then
            result = FPZero(sign, M);
        elsif fpcr.DN == '1' then
            result = FPDefaultNaN(fpcr, M);
        else
            result = FPConvertNaN(op, M);
        if fptype == FPType_SNaN || alt_hp then
            FPProcessException(FPExc_InvalidOp,fpcr);
    elsif fptype == FPType_Infinity then
        if alt_hp then
            result = sign:Ones(M-1);
            FPProcessException(FPExc_InvalidOp, fpcr);
        else
            result = FPInfinity(sign, M);
    elsif fptype == FPType_Zero then
        result = FPZero(sign, M);
    else
        result = FPRoundCV(value, fpcr, rounding, M);
    return result;

// FPConvert()
// ===========

bits(M) FPConvert(bits(N) op, FPCR_Type fpcr, integer M)
    return FPConvert(op, fpcr, FPRoundingMode(fpcr), M);
```

G1.2.3.149 FPConvertNaN

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// FPConvertNaN()
// ==============
// Converts a NaN of one floating-point type to another

bits(M) FPConvertNaN(bits(N) op, integer M)
    assert N IN {16,32,64};
    assert M IN {16,32,64};
    bits(M) result;
    bits(51) frac;

    sign = op<N-1>;

    // Unpack payload from input NaN
    case N of
        when 64 frac = op<50:0>;
        when 32 frac = op<21:0>:Zeros(29);
        when 16 frac = op<8:0>:Zeros(42);

    // Repack payload into output NaN, while
    // converting an SNaN to a QNaN.
    case M of
        when 64 result = sign:Ones(M-52):frac;
        when 32 result = sign:Ones(M-23):frac<50:29>;
        when 16 result = sign:Ones(M-10):frac<50:42>;

    return result;
```

G1.2.3.150 FPDecodeRM

```
// FPDecodeRM()
// ============

// Decode most common AArch32 floating-point rounding encoding.

FPRounding FPDecodeRM(bits(2) rm)
    FPRounding result;
    case rm of
        when '00' result = FPRounding_TIEAWAY; // A
        when '01' result = FPRounding_TIEEVEN; // N
        when '10' result = FPRounding_POSINF;  // P
        when '11' result = FPRounding_NEGINF;  // M

    return result;
```

G1.2.3.151 FPDecodeRounding

```
// FPDecodeRounding()
// ==================

// Decode floating-point rounding mode and common AArch64 encoding.

FPRounding FPDecodeRounding(bits(2) rmode)
    case rmode of
        when '00' return FPRounding_TIEEVEN; // N
        when '01' return FPRounding_POSINF;  // P
        when '10' return FPRounding_NEGINF;  // M
        when '11' return FPRounding_ZERO;    // Z
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.3.152 FPDefaultNaN

```
// FPDefaultNaN()
// ==============

bits(N) FPDefaultNaN(FPCR_Type fpcr, integer N)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 else (if N == 32 then 8 else 11));
    constant integer F = N - (E + 1);
    constant bit sign = '0';

    constant bits(E) exp  = Ones(E);
    constant bits(F) frac = '1':Zeros(F-1);
    return sign : exp : frac;
```

G1.2.3.153 FPDiv

```
// FPDiv()
// =======

bits(N) FPDiv(bits(N) op1, bits(N) op2, FPCR_Type fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);

    if !done then
        inf1  = type1 == FPType_Infinity;
        inf2  = type2 == FPType_Infinity;
        zero1 = type1 == FPType_Zero;
        zero2 = type2 == FPType_Zero;

        if (inf1 && inf2) || (zero1 && zero2) then
            result = FPDefaultNaN(fpcr, N);
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif inf1 || zero2 then
            result = FPInfinity(sign1 EOR sign2, N);
            if !inf1 then FPProcessException(FPExc_DivideByZero, fpcr);
        elsif zero1 || inf2 then
            result = FPZero(sign1 EOR sign2, N);
        else
            result = FPRound(value1/value2, fpcr, N);

    return result;
```

G1.2.3.154 FPExc

```
// FPExc
// =====

enumeration FPExc       {FPExc_InvalidOp, FPExc_DivideByZero, FPExc_Overflow,
                         FPExc_Underflow, FPExc_Inexact, FPExc_InputDenorm};
```

G1.2.3.155 FPInfinity

```
// FPInfinity()
// ============
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
bits(N) FPInfinity(bit sign, integer N)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 else (if N == 32 then 8 else 11));
    constant integer F = N - (E + 1);
    constant bits(E) exp  = Ones(E);
    constant bits(F) frac = Zeros(F);
    return sign : exp : frac;
```

G1.2.3.156 FPMax

```
// FPMax()
// =======
// Compare two inputs and return the greater value after rounding. The
// 'fpcr' argument supplies the FPCR control bits.

bits(N) FPMax(bits(N) op1, bits(N) op2, FPCR_Type fpcr_in)
    assert N IN {16,32,64};
    boolean done;
    bits(N) result;
    FPCR_Type fpcr = fpcr_in;
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        FPType fptype;
        bit sign;
        real value;
        if value1 > value2 then
            (fptype,sign,value) = (type1,sign1,value1);
        else
            (fptype,sign,value) = (type2,sign2,value2);
        if fptype == FPType_Infinity then
            result = FPInfinity(sign, N);
        elsif fptype == FPType_Zero then
            sign = sign1 AND sign2;         // Use most positive sign
            result = FPZero(sign, N);
        else
            // The use of FPRound() covers the case where there is a trapped underflow exception
            // for a denormalized number even though the result is exact.
            rounding = FPRoundingMode(fpcr);
            result = FPRound(value, fpcr, rounding, N);

    return result;
```

G1.2.3.157 FPMaxNormal

```
// FPMaxNormal()
// =============

bits(N) FPMaxNormal(bit sign, integer N)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 else (if N == 32 then 8 else 11));
    constant integer F = N - (E + 1);
    exp  = Ones(E-1):'0';
    frac = Ones(F);
    return sign : exp : frac;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.3.158 FPMaxNum

```
// FPMaxNum()
// ==========

bits(N) FPMaxNum(bits(N) op1_in, bits(N) op2_in, FPCR_Type fpcr)
    assert N IN {16,32,64};
    bits(N) op1 = op1_in;
    bits(N) op2 = op2_in;
    (type1,-,-) = FPUnpack(op1, fpcr);
    (type2,-,-) = FPUnpack(op2, fpcr);

    // Treat a single quiet-NaN as -Infinity.
    if type1 == FPType_QNaN && type2 != FPType_QNaN then
        op1 = FPInfinity('1', N);
    elsif type1 != FPType_QNaN && type2 == FPType_QNaN then
        op2 = FPInfinity('1', N);
    result = FPMax(op1, op2, fpcr);

    return result;
```

G1.2.3.159 IsMerging

```
// IsMerging()
// ===========
// Returns TRUE if the output elements other than the lowest are taken from
// the destination register.

boolean IsMerging(FPCR_Type fpcr)
    return FALSE;
```

G1.2.3.160 FPMin

```
// FPMin()
// =======
// Compare two operands and return the smaller operand after rounding. The
// 'fpcr' argument supplies the FPCR control bits.

bits(N) FPMin(bits(N) op1, bits(N) op2, FPCR_Type fpcr_in)
    assert N IN {16,32,64};
    boolean done;
    bits(N) result;
    FPCR_Type fpcr = fpcr_in;
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        FPType fptype;
        bit sign;
        real value;
        FPRounding rounding;
        if value1 < value2 then
            (fptype,sign,value) = (type1,sign1,value1);
        else
            (fptype,sign,value) = (type2,sign2,value2);
        if fptype == FPType_Infinity then
            result = FPInfinity(sign, N);
        elsif fptype == FPType_Zero then
            sign = sign1 OR sign2;              // Use most negative sign
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            result = FPZero(sign, N);
        else
            // The use of FPRound() covers the case where there is a trapped underflow exception
            // for a denormalized number even though the result is exact.
            rounding = FPRoundingMode(fpcr);
            result = FPRound(value, fpcr, rounding, N);
    return result;
```

G1.2.3.161 FPMinNum

```
// FPMinNum()
// ==========

bits(N) FPMinNum(bits(N) op1_in, bits(N) op2_in, FPCR_Type fpcr)
    assert N IN {16,32,64};
    bits(N) op1 = op1_in;
    bits(N) op2 = op2_in;
    (type1,-,-) = FPUnpack(op1, fpcr);
    (type2,-,-) = FPUnpack(op2, fpcr);

    // Treat a single quiet-NaN as +Infinity.
    if type1 == FPType_QNaN && type2 != FPType_QNaN then
        op1 = FPInfinity('0', N);
    elsif type1 != FPType_QNaN && type2 == FPType_QNaN then
        op2 = FPInfinity('0', N);
    result = FPMin(op1, op2, fpcr);

    return result;
```

G1.2.3.162 FPMul

```
// FPMul()
// =======

bits(N) FPMul(bits(N) op1, bits(N) op2, FPCR_Type fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);

        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPDefaultNaN(fpcr, N);
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2, N);
        elsif zero1 || zero2 then
            result = FPZero(sign1 EOR sign2, N);
        else
            result = FPRound(value1*value2, fpcr, N);

    return result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.3.163 FPMulAdd

```
// FPMulAdd()
// ==========
//
// Calculates addend + op1*op2 with a single rounding. The 'fpcr' argument
// supplies the FPCR control bits.

bits(N) FPMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, FPCR_Type fpcr)
    assert N IN {16,32,64};

    (typeA,signA,valueA) = FPUnpack(addend, fpcr);
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    rounding = FPRoundingMode(fpcr);
    inf1 = (type1 == FPType_Infinity); zero1 = (type1 == FPType_Zero);
    inf2 = (type2 == FPType_Infinity); zero2 = (type2 == FPType_Zero);

    (done,result) = FPProcessNaNs3(typeA, type1, type2, addend, op1, op2, fpcr);

    if typeA == FPType_QNaN && ((inf1 && zero2) || (zero1 && inf2)) then
        result = FPDefaultNaN(fpcr, N);
        FPProcessException(FPExc_InvalidOp, fpcr);

    if !done then
        infA = (typeA == FPType_Infinity); zeroA = (typeA == FPType_Zero);

        // Determine sign and type product will have if it does not cause an
        // Invalid Operation.
        signP = sign1 EOR sign2;
        infP  = inf1 || inf2;
        zeroP = zero1 || zero2;

        // Non SNaN-generated Invalid Operation cases are multiplies of zero
        // by infinity and additions of opposite-signed infinities.
        invalidop = (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP);

        if invalidop then
            result = FPDefaultNaN(fpcr, N);
            FPProcessException(FPExc_InvalidOp, fpcr);
        // Other cases involving infinities produce an infinity of the same sign.
        elsif (infA && signA == '0') || (infP && signP == '0') then
            result = FPInfinity('0', N);
        elsif (infA && signA == '1') || (infP && signP == '1') then
            result = FPInfinity('1', N);

        // Cases where the result is exactly zero and its sign is not determined by the
        // rounding mode are additions of same-signed zeros.
        elsif zeroA && zeroP && signA == signP then
            result = FPZero(signA, N);

        // Otherwise calculate numerical result and round it.
        else
            result_value = valueA + (value1 * value2);
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign, N);
            else
                result = FPRound(result_value, fpcr, N);
    return result;
```

G1.2.3.164 FPMulAddH

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// FPMulAddH()
// ===========
// Calculates addend + op1*op2.

bits(32) FPMulAddH(bits(32) addend, bits(16) op1, bits(16) op2, FPCR_Type fpcr)
    rounding = FPRoundingMode(fpcr);
    (typeA,signA,valueA) = FPUnpack(addend, fpcr);
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    inf1 = (type1 == FPType_Infinity); zero1 = (type1 == FPType_Zero);
    inf2 = (type2 == FPType_Infinity); zero2 = (type2 == FPType_Zero);

    (done,result) = FPProcessNaNs3H(typeA, type1, type2, addend, op1, op2, fpcr);

    if typeA == FPType_QNaN && ((inf1 && zero2) || (zero1 && inf2)) then
        result = FPDefaultNaN(fpcr, 32);
        FPProcessException(FPExc_InvalidOp, fpcr);

    if !done then
        infA = (typeA == FPType_Infinity); zeroA = (typeA == FPType_Zero);

        // Determine sign and type product will have if it does not cause an
        // Invalid Operation.
        signP = sign1 EOR sign2;
        infP = inf1 || inf2;
        zeroP = zero1 || zero2;

        // Non SNaN-generated Invalid Operation cases are multiplies of zero by infinity and
        // additions of opposite-signed infinities.
        invalidop = (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP);

        if invalidop then
            result = FPDefaultNaN(fpcr, 32);
            FPProcessException(FPExc_InvalidOp, fpcr);

        // Other cases involving infinities produce an infinity of the same sign.
        elsif (infA && signA == '0') || (infP && signP == '0') then
            result = FPInfinity('0', 32);
        elsif (infA && signA == '1') || (infP && signP == '1') then
            result = FPInfinity('1', 32);

        // Cases where the result is exactly zero and its sign is not determined by the
        // rounding mode are additions of same-signed zeros.
        elsif zeroA && zeroP && signA == signP then
            result = FPZero(signA, 32);

        // Otherwise calculate numerical result and round it.
        else
            result_value = valueA + (value1 * value2);
            if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign, 32);
            else
                result = FPRound(result_value, fpcr, 32);

    return result;
```

G1.2.3.165 FPProcessNaNs3H

```
// FPProcessNaNs3H()
// =================

(boolean, bits(32)) FPProcessNaNs3H(FPType type1, FPType type2, FPType type3,
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                                    bits(32) op1, bits(16) op2, bits(16) op3,
                                    FPCR_Type fpcr)

    bits(32) result;
    boolean done;
    if type1 == FPType_SNaN then
        done = TRUE; result = FPProcessNaN(type1, op1, fpcr);
    elsif type2 == FPType_SNaN then
        done = TRUE; result = FPConvertNaN(FPProcessNaN(type2, op2, fpcr), 32);
    elsif type3 == FPType_SNaN then
        done = TRUE; result = FPConvertNaN(FPProcessNaN(type3, op3, fpcr), 32);
    elsif type1 == FPType_QNaN then
        done = TRUE; result = FPProcessNaN(type1, op1, fpcr);
    elsif type2 == FPType_QNaN then
        done = TRUE; result = FPConvertNaN(FPProcessNaN(type2, op2, fpcr), 32);
    elsif type3 == FPType_QNaN then
        done = TRUE; result = FPConvertNaN(FPProcessNaN(type3, op3, fpcr), 32);
    else
        done = FALSE; result = Zeros(32); // 'Don't care' result
    return (done, result);
```

G1.2.3.166 FPMulX

```
// FPMulX()
// ========

bits(N) FPMulX(bits(N) op1, bits(N) op2, FPCR_Type fpcr)
    assert N IN {16,32,64};
    bits(N) result;
    boolean done;
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);

        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPTwo(sign1 EOR sign2, N);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2, N);
        elsif zero1 || zero2 then
            result = FPZero(sign1 EOR sign2, N);
        else
            result = FPRound(value1*value2, fpcr, N);

    return result;
```

G1.2.3.167 FPNeg

```
// FPNeg()
// =======

bits(N) FPNeg(bits(N) op, FPCR_Type fpcr)
    assert N IN {16,32,64};
    return NOT(op<N-1>) : op<N-2:0>;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.3.168 FPOnePointFive

```
// FPOnePointFive()
// ================

bits(N) FPOnePointFive(bit sign, integer N)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 else (if N == 32 then 8 else 11));
    constant integer F = N - (E + 1);
    exp  = '0':Ones(E-1);
    frac = '1':Zeros(F-1);
    result = sign : exp : frac;

    return result;
```

G1.2.3.169 FPProcessException

```
// FPProcessException()
// ====================
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

FPProcessException(FPExc except, FPCR_Type fpcr)
    integer cumul;
    // Determine the cumulative exception bit number
    case except of
        when FPExc_InvalidOp     cumul = 0;
        when FPExc_DivideByZero  cumul = 1;
        when FPExc_Overflow      cumul = 2;
        when FPExc_Underflow     cumul = 3;
        when FPExc_Inexact       cumul = 4;
        when FPExc_InputDenorm   cumul = 7;
    enable = cumul + 8;
    if fpcr<enable> == '1' then
        // Trapping of the exception enabled.
        // It is IMPLEMENTATION DEFINED whether the enable bit may be set at all,
        // and if so then how exceptions and in what order that they may be
        // accumulated before calling FPTrappedException().
        bits(8) accumulated_exceptions = GetAccumulatedFPExceptions();
        accumulated_exceptions<cumul> = '1';
        if boolean IMPLEMENTATION_DEFINED "Support trapping of floating-point exceptions" then
            is_ase = IsASEInstruction();
            AArch64.FPTrappedException(is_ase, accumulated_exceptions);
        else
            // The exceptions generated by this instruction are accumulated by the PE and
            // FPTrappedException is called later during its execution, before the next
            // instruction is executed. This field is cleared at the start of each FP instruction.
            SetAccumulatedFPExceptions(accumulated_exceptions);
    else
        // Set the cumulative exception bit
        FPSR<cumul> = '1';
    return;
```

G1.2.3.170 FPProcessNaN

```
// FPProcessNaN()
// ==============
// Handle NaN input operands, returning the operand or default NaN value
// if fpcr.DN is selected. The 'fpcr' argument supplies the FPCR control bits.
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
bits(N) FPProcessNaN(FPType fptype, bits(N) op, FPCR_Type fpcr)
    assert N IN {16,32,64};
    assert fptype IN {FPType_QNaN, FPType_SNaN};
    integer topfrac;

    case N of
        when 16 topfrac =  9;
        when 32 topfrac = 22;
        when 64 topfrac = 51;

    result = op;
    if fptype == FPType_SNaN then
        result<topfrac> = '1';
        FPProcessException(FPExc_InvalidOp, fpcr);
    if fpcr.DN == '1' then  // DefaultNaN requested
        result = FPDefaultNaN(fpcr, N);
    return result;
```

G1.2.3.171 FPProcessNaNs

```
// FPProcessNaNs()
// ===============
//
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

(boolean, bits(N)) FPProcessNaNs(FPType type1, FPType type2, bits(N) op1,
                                 bits(N) op2, FPCR_Type fpcr)
    assert N IN {16,32,64};
    boolean done;
    bits(N) result;
    if type1 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type1, op1, fpcr);
    elsif type2 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type2, op2, fpcr);
    elsif type1 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type1, op1, fpcr);
    elsif type2 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type2, op2, fpcr);
    else
        done = FALSE;  result = Zeros(N);  // 'Don't care' result
    return (done, result);
```

G1.2.3.172 FPProcessNaNs3

```
// FPProcessNaNs3()
// ================
//
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
(boolean, bits(N)) FPProcessNaNs3(FPType type1, FPType type2, FPType type3,
                                  bits(N) op1, bits(N) op2, bits(N) op3,
                                  FPCR_Type fpcr)
    assert N IN {16,32,64};
    bits(N) result;
    boolean done;

    if type1 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type1, op1, fpcr);
    elsif type2 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type2, op2, fpcr);
    elsif type3 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type3, op3, fpcr);
    elsif type1 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type1, op1, fpcr);
    elsif type2 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type2, op2, fpcr);
    elsif type3 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type3, op3, fpcr);
    else
        done = FALSE;  result = Zeros(N);  // 'Don't care' result
    return (done, result);
```

G1.2.3.173 FPRecipEstimate

```
// FPRecipEstimate()
// =================

bits(N) FPRecipEstimate(bits(N) operand, FPCR_Type fpcr_in)
    assert N IN {16,32,64};
    FPCR_Type fpcr = fpcr_in;
    bits(N) result;
    boolean overflow_to_inf;
    (fptype,sign,value) = FPUnpack(operand, fpcr);

    constant FPRounding rounding = FPRoundingMode(fpcr);
    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FPProcessNaN(fptype, operand, fpcr);
    elsif fptype == FPType_Infinity then
        result = FPZero(sign, N);
    elsif fptype == FPType_Zero then
        result = FPInfinity(sign, N);
        FPProcessException(FPExc_DivideByZero, fpcr);
    elsif (
            (N == 16 && Abs(value) < 2.0^-16) ||
            (N == 32 && Abs(value) < 2.0^-128) ||
            (N == 64 && Abs(value) < 2.0^-1024)
          ) then
        case rounding of
            when FPRounding_TIEEVEN
                overflow_to_inf = TRUE;
            when FPRounding_POSINF
                overflow_to_inf = (sign == '0');
            when FPRounding_NEGINF
                overflow_to_inf = (sign == '1');
            when FPRounding_ZERO
                overflow_to_inf = FALSE;
        result = if overflow_to_inf then FPInfinity(sign, N) else FPMaxNormal(sign, N);
        FPProcessException(FPExc_Overflow, fpcr);
        FPProcessException(FPExc_Inexact, fpcr);
    elsif ((fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16))
          && (
                (N == 16 && Abs(value) >= 2.0^14) ||
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            (N == 32 && Abs(value) >= 2.0^126) ||
            (N == 64 && Abs(value) >= 2.0^1022)
          ) then
        // Result flushed to zero of correct sign
        result = FPZero(sign, N);

        // Flush-to-zero never generates a trapped exception.
        FPSR.UFC = '1';
    else
        // Scale to a fixed point value in the range 0.5 <= x < 1.0 in steps of 1/512, and
        // calculate result exponent. Scaled value has copied sign bit,
        // exponent = 1022 = double-precision biased version of -1,
        // fraction = original fraction
        bits(52) fraction;
        integer exp;
        case N of
            when 16
                fraction = operand<9:0> : Zeros(42);
                exp = UInt(operand<14:10>);
            when 32
                fraction = operand<22:0> : Zeros(29);
                exp = UInt(operand<30:23>);
            when 64
                fraction = operand<51:0>;
                exp = UInt(operand<62:52>);

        if exp == 0 then
            if fraction<51> == '0' then
                exp = -1;
                fraction = fraction<49:0>:'00';
            else
                fraction = fraction<50:0>:'0';

        constant integer scaled = UInt('1':fraction<51:44>);

        integer result_exp;
        case N of
            when 16 result_exp =   29 - exp; // In range 29-30 = -1 to 29+1 = 30
            when 32 result_exp =  253 - exp; // In range 253-254 = -1 to 253+1 = 254
            when 64 result_exp = 2045 - exp; // In range 2045-2046 = -1 to 2045+1 = 2046

        // Scaled is in range 256 .. 511 representing a fixed-point number in range [0.5 .. 1.0].
        estimate = RecipEstimate(scaled);

        // Estimate is in the range 256 .. 511 representing a fixed-point
        // result in the range [1.0 .. 2.0].
        // Convert to scaled floating point result with copied sign bit,
        // high-order bits from estimate, and exponent calculated above.
        fraction = estimate<7:0> : Zeros(44);

        if result_exp == 0 then
            fraction = '1' : fraction<51:1>;
        elsif result_exp == -1 then
            fraction = '01' : fraction<51:2>;
            result_exp = 0;

        case N of
            when 16 result = sign : result_exp<N-12:0> : fraction<51:42>;
            when 32 result = sign : result_exp<N-25:0> : fraction<51:29>;
            when 64 result = sign : result_exp<N-54:0> : fraction<51:0>;

    return result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.3.174 RecipEstimate

```
// RecipEstimate()
// ===============
// Compute estimate of reciprocal of 9-bit fixed-point number.
//
// a is in range 256 .. 511 representing a number in the range 0.5 <= x < 1.0.
// result is in the range 256 .. 511 representing a number in the range 1.0 to 511/256.

integer RecipEstimate(integer a_in)
    integer a = a_in;
    assert 256 <= a && a < 512;
    a = a*2+1;          // Round to nearest
    constant integer b = (2 ^ 19) DIV a;
    r = (b+1) DIV 2; // Round to nearest
    assert 256 <= r && r < 512;
    return r;
```

G1.2.3.175 FPRecpX

```
// FPRecpX()
// =========

bits(N) FPRecpX(bits(N) op, FPCR_Type fpcr_in)
    assert N IN {16,32,64};
    FPCR_Type fpcr = fpcr_in;
    constant (F, -) = FPBits(N);
    constant E = (N - F) - 1;
    bits(N)           result;
    bits(E)           exp;
    bits(E)           max_exp;
    constant bits(F) frac = Zeros(F);

    (fptype,sign,value) = FPUnpack(op, fpcr);
    exp = op<F+:E>;
    max_exp = Ones(E) - 1;

    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FPProcessNaN(fptype, op, fpcr);
    else
        if IsZero(exp) then                 // Zero and denormals
            result = ZeroExtend(sign:max_exp:frac, N);
        else                                // Infinities and normals
            result = ZeroExtend(sign:NOT(exp):frac, N);

    return result;
```

G1.2.3.176 FPRound

```
// FPRound()
// =========
// Generic conversion from precise, unbounded real data type to IEEE format.

bits(N) FPRound(real op, FPCR_Type fpcr, integer N)
    return FPRound(op, fpcr, FPRoundingMode(fpcr), N);

// FPRound()
// =========
// For directed FP conversion, includes an explicit 'rounding' argument.
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
bits(N) FPRound(real op, FPCR_Type fpcr_in, FPRounding rounding, integer N)
    FPCR_Type fpcr = fpcr_in;
    fpcr.AHP = '0';
    return FPRoundBase(op, fpcr, rounding, N);
```

G1.2.3.177 FPRoundBase

```
// FPRoundBase()
// =============
// Convert a real number 'op' into an N-bit floating-point value using the
// supplied rounding mode 'rounding'.

bits(N) FPRoundBase(real op, FPCR_Type fpcr, FPRounding rounding, integer N)
    assert N IN {16,32,64};
    assert op != 0.0;
    assert rounding != FPRounding_TIEAWAY;
    bits(N) result;

    // Obtain format parameters - minimum exponent, numbers of exponent and fraction bits.
    constant (F, minimum_exp) = FPBits(N);
    constant E = (N - F) - 1;
    // Split value into sign, unrounded mantissa and exponent.
    bit sign;
    integer exponent;
    real mantissa;
    if op < 0.0 then
        sign = '1';  mantissa = -op;
    else
        sign = '0';  mantissa = op;
    (mantissa, exponent) = NormalizeReal(mantissa);

    if (((fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16)) &&
        exponent < minimum_exp) then
        // Flush-to-zero never generates a trapped exception.
        FPSR.UFC = '1';
        return FPZero(sign, N);

    // Start creating the exponent value for the result. Start by biasing the actual exponent
    // so that the minimum exponent becomes 1, lower values 0 (indicating possible underflow).
    biased_exp = Max((exponent - minimum_exp) + 1, 0);
    if biased_exp == 0 then mantissa = mantissa / 2.0^(minimum_exp - exponent);

    // Get the unrounded mantissa as an integer, and the "units in last place" rounding error.
    int_mant = RoundDown(mantissa * 2.0^F);  // < 2.0^F if biased_exp == 0, >= 2.0^F if not
    error = mantissa * 2.0^F - Real(int_mant);

    // Underflow occurs if exponent is too small before rounding, and result is inexact or
    // the Underflow exception is trapped.
    constant boolean trapped_UF = fpcr.UFE == '1';
    if biased_exp == 0 && (error != 0.0 || trapped_UF) then
        FPProcessException(FPExc_Underflow, fpcr);

    // Round result according to rounding mode.
    boolean round_up;
    boolean overflow_to_inf;
    case rounding of
        when FPRounding_TIEEVEN
            round_up = (error > 0.5 || (error == 0.5 && int_mant<0> == '1'));
            overflow_to_inf = TRUE;
        when FPRounding_POSINF
            round_up = (error != 0.0 && sign == '0');
            overflow_to_inf = (sign == '0');
        when FPRounding_NEGINF
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            round_up = (error != 0.0 && sign == '1');
            overflow_to_inf = (sign == '1');
        when FPRounding_ZERO, FPRounding_ODD
            round_up = FALSE;
            overflow_to_inf = FALSE;

    if round_up then
        int_mant = int_mant + 1;
        if int_mant == 2^F then      // Rounded up from denormalized to normalized
            biased_exp = 1;
        if int_mant == 2^(F+1) then  // Rounded up to next exponent
            biased_exp = biased_exp + 1;
            int_mant = int_mant DIV 2;

    // Handle rounding to odd
    if error != 0.0 && rounding == FPRounding_ODD && int_mant<0> == '0' then
        int_mant = int_mant + 1;

    // Deal with overflow and generate result.
    if N != 16 || fpcr.AHP == '0' then  // Single, double or IEEE half precision
        if biased_exp >= 2^E - 1 then
            result = if overflow_to_inf then FPInfinity(sign, N) else FPMaxNormal(sign, N);
            FPProcessException(FPExc_Overflow, fpcr);
            error = 1.0;  // Ensure that an Inexact exception occurs
        else
            result = sign : biased_exp<E-1:0> : int_mant<F-1:0> : Zeros(N-(E+F+1));
    else                                    // Alternative half precision
        if biased_exp >= 2^E then
            result = sign : Ones(N-1);
            FPProcessException(FPExc_InvalidOp, fpcr);
            error = 0.0;  // Ensure that an Inexact exception does not occur
        else
            result = sign : biased_exp<E-1:0> : int_mant<F-1:0> : Zeros(N-(E+F+1));

    // Deal with Inexact exception.
    if error != 0.0 then
        FPProcessException(FPExc_Inexact, fpcr);

    return result;
```

G1.2.3.178 FPRoundCV

```
// FPRoundCV()
// ===========
// Used for FP to FP conversion instructions.
// For half-precision data ignores FZ16 and observes AHP.

bits(N) FPRoundCV(real op, FPCR_Type fpcr_in, FPRounding rounding, integer N)
    FPCR_Type fpcr = fpcr_in;
    fpcr.FZ16 = '0';
    return FPRoundBase(op, fpcr, rounding, N);
```

G1.2.3.179 FPRounding

```
// FPRounding
// ==========
// The conversion and rounding functions take an explicit
// rounding mode enumeration instead of booleans or FPCR values.

enumeration FPRounding  {FPRounding_TIEEVEN, FPRounding_POSINF,
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                        FPRounding_NEGINF,  FPRounding_ZERO,
                        FPRounding_TIEAWAY, FPRounding_ODD};
```

G1.2.3.180 FPRoundingMode

```
// FPRoundingMode()
// ================
// Return the current floating-point rounding mode.

FPRounding FPRoundingMode(FPCR_Type fpcr)
    return FPDecodeRounding(fpcr.RMode);
```

G1.2.3.181 FPRoundInt

```
// FPRoundInt()
// ============

// Round op to nearest integral floating point value using rounding mode in FPCR/FPSCR.
// If EXACT is TRUE, set FPSR.IXC if result is not numerically equal to op.

bits(N) FPRoundInt(bits(N) op, FPCR_Type fpcr, FPRounding rounding, boolean exact)
    assert rounding != FPRounding_ODD;
    assert N IN {16,32,64};

    // Unpack using FPCR to determine if subnormals are flushed-to-zero.
    (fptype,sign,value) = FPUnpack(op, fpcr);

    bits(N) result;
    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FPProcessNaN(fptype, op, fpcr);
    elsif fptype == FPType_Infinity then
        result = FPInfinity(sign, N);
    elsif fptype == FPType_Zero then
        result = FPZero(sign, N);
    else
        // Extract integer component.
        int_result = RoundDown(value);
        error = value - Real(int_result);

        // Determine whether supplied rounding mode requires an increment.
        boolean round_up;
        case rounding of
            when FPRounding_TIEEVEN
                round_up = (error > 0.5 || (error == 0.5 && int_result<0> == '1'));
            when FPRounding_POSINF
                round_up = (error != 0.0);
            when FPRounding_NEGINF
                round_up = FALSE;
            when FPRounding_ZERO
                round_up = (error != 0.0 && int_result < 0);
            when FPRounding_TIEAWAY
                round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));

        if round_up then int_result = int_result + 1;

        // Convert integer value into an equivalent real value.
        real_result = Real(int_result);

        // Re-encode as a floating-point value, result is always exact.
        if real_result == 0.0 then
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            result = FPZero(sign, N);
        else
            result = FPRound(real_result, fpcr, FPRounding_ZERO, N);

        // Generate inexact exceptions.
        if error != 0.0 && exact then
            FPProcessException(FPExc_Inexact, fpcr);

    return result;
```

G1.2.3.182 FPRSqrtEstimate

```
// FPRSqrtEstimate()
// =================

bits(N) FPRSqrtEstimate(bits(N) operand, FPCR_Type fpcr_in)
    assert N IN {16,32,64};
    FPCR_Type fpcr = fpcr_in;

    (fptype,sign,value) = FPUnpack(operand, fpcr);

    bits(N) result;
    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FPProcessNaN(fptype, operand, fpcr);
    elsif fptype == FPType_Zero then
        result = FPInfinity(sign, N);
        FPProcessException(FPExc_DivideByZero, fpcr);
    elsif sign == '1' then
        result = FPDefaultNaN(fpcr, N);
        FPProcessException(FPExc_InvalidOp, fpcr);
    elsif fptype == FPType_Infinity then
        result = FPZero('0', N);
    else
        // Scale to a fixed-point value in the range 0.25 <= x < 1.0 in steps of 512, with the
        // evenness or oddness of the exponent unchanged, and calculate result exponent.
        // Scaled value has copied sign bit, exponent = 1022 or 1021 = double-precision
        // biased version of -1 or -2, fraction = original fraction extended with zeros.

        bits(52) fraction;
        integer exp;
        case N of
            when 16
                fraction = operand<9:0> : Zeros(42);
                exp = UInt(operand<14:10>);
            when 32
                fraction = operand<22:0> : Zeros(29);
                exp = UInt(operand<30:23>);
            when 64
                fraction = operand<51:0>;
                exp = UInt(operand<62:52>);

        if exp == 0 then
            while fraction<51> == '0' do
                fraction = fraction<50:0> : '0';
                exp = exp - 1;
            fraction = fraction<50:0> : '0';

        integer scaled;
        if exp<0> == '0' then
            scaled = UInt('1':fraction<51:44>);
        else
            scaled = UInt('01':fraction<51:45>);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        integer result_exp;
        case N of
            when 16 result_exp = (  44 - exp) DIV 2;
            when 32 result_exp = ( 380 - exp) DIV 2;
            when 64 result_exp = (3068 - exp) DIV 2;

        estimate = RecipSqrtEstimate(scaled);

        // Estimate is in the range 256 .. 511 representing a fixed point
        // result in the range [1.0 .. 2.0].
        // Convert to scaled floating point result with copied sign bit and high-order
        // fraction bits, and exponent calculated above.
        case N of
            when 16 result = '0' : result_exp<N-12:0> : estimate<7:0>:Zeros(2);
            when 32 result = '0' : result_exp<N-25:0> : estimate<7:0>:Zeros(15);
            when 64 result = '0' : result_exp<N-54:0> : estimate<7:0>:Zeros(44);

    return result;
```

G1.2.3.183 RecipSqrtEstimate

```
// RecipSqrtEstimate()
// ===================
// Compute estimate of reciprocal square root of 9-bit fixed-point number.
//
// a_in is in range 128 .. 511 representing a number in the range 0.25 <= x < 1.0.
// result is in the range 256 .. 511 representing a number in the range in the range 1.0 to 511/256.

integer RecipSqrtEstimate(integer a_in)
    integer a = a_in;
    assert 128 <= a && a < 512;
    if a < 256 then             // 0.25 .. 0.5
        a = a*2+1;              // a in units of 1/512 rounded to nearest
    else // 0.5 .. 1.0
        a = (a >> 1) << 1;      // Discard bottom bit
        a = (a+1)*2;            // a in units of 1/256 rounded to nearest
    integer b = 512;
    while a*(b+1)*(b+1) < 2^28 do
        b = b+1;                // b = largest b such that b < 2^14 / sqrt(a)
    r = (b+1) DIV 2;          // Round to nearest
    assert 256 <= r && r < 512;
    return r;
```

G1.2.3.184 FPSqrt

```
// FPSqrt()
// ========

bits(N) FPSqrt(bits(N) op, FPCR_Type fpcr)
    assert N IN {16,32,64};
    (fptype,sign,value) = FPUnpack(op, fpcr);

    bits(N) result;
    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FPProcessNaN(fptype, op, fpcr);
    elsif fptype == FPType_Zero then
        result = FPZero(sign, N);
    elsif fptype == FPType_Infinity && sign == '0' then
        result = FPInfinity(sign, N);
    elsif sign == '1' then
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        result = FPDefaultNaN(fpcr, N);
        FPProcessException(FPExc_InvalidOp, fpcr);
    else
        integer prec;
        boolean inexact;
        if N == 16 then
            prec = 13; // 10 fraction bit + 3
        elsif N == 32 then
            prec = 26; // 23 fraction bits + 3
        else // N == 64
            prec = 55; // 52 fraction bits + 3
        (value, inexact) = SqrtRoundDown(value, prec);
        result = FPRound(value, fpcr, N);
        if inexact then
            FPProcessException(FPExc_Inexact, fpcr);
    return result;
```

G1.2.3.185 FPSub

```
// FPSub()
// =======

bits(N) FPSub(bits(N) op1, bits(N) op2, FPCR_Type fpcr)

    assert N IN {16,32,64};
    rounding = FPRoundingMode(fpcr);

    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);

        if inf1 && inf2 && sign1 == sign2 then
            result = FPDefaultNaN(fpcr, N);
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
            result = FPInfinity('0', N);
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
            result = FPInfinity('1', N);
        elsif zero1 && zero2 && sign1 == NOT(sign2) then
            result = FPZero(sign1, N);
        else
            result_value = value1 - value2;
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign, N);
            else
                result = FPRound(result_value, fpcr, rounding, N);

    return result;
```

G1.2.3.186 FPThree

```
// FPThree()
// =========
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
bits(N) FPThree(bit sign, integer N)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 else (if N == 32 then 8 else 11));
    constant integer F = N - (E + 1);
    exp  = '1':Zeros(E-1);
    frac = '1':Zeros(F-1);
    result = sign : exp : frac;

    return result;
```

G1.2.3.187 FPToFixed

```
// FPToFixed()
// ===========

// Convert N-bit precision floating point 'op' to M-bit fixed point with
// FBITS fractional bits, controlled by UNSIGNED and ROUNDING.

bits(M) FPToFixed(bits(N) op, integer fbits, boolean unsigned, FPCR_Type fpcr,
                  FPRounding rounding, integer M)
    assert N IN {16,32,64};
    assert M IN {16,32,64};
    assert fbits >= 0;
    assert rounding != FPRounding_ODD;

    // Unpack using fpcr to determine if subnormals are flushed-to-zero.
    (fptype,sign,value) = FPUnpack(op, fpcr);

    // If NaN, set cumulative flag or take exception.
    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        FPProcessException(FPExc_InvalidOp, fpcr);

    // Scale by fractional bits and produce integer rounded towards minus-infinity.
    value = value * 2.0^fbits;
    int_result = RoundDown(value);
    error = value - Real(int_result);

    // Determine whether supplied rounding mode requires an increment.
    boolean round_up;
    case rounding of
        when FPRounding_TIEEVEN
            round_up = (error > 0.5 || (error == 0.5 && int_result<0> == '1'));
        when FPRounding_POSINF
            round_up = (error != 0.0);
        when FPRounding_NEGINF
            round_up = FALSE;
        when FPRounding_ZERO
            round_up = (error != 0.0 && int_result < 0);
        when FPRounding_TIEAWAY
            round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));

    if round_up then int_result = int_result + 1;

    // Generate saturated result and exceptions.
    (result, overflow) = SatQ(int_result, M, unsigned);
    if overflow then
        FPProcessException(FPExc_InvalidOp, fpcr);
    elsif error != 0.0 then
        FPProcessException(FPExc_Inexact, fpcr);

    return result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.3.188 FPToFixedJS

```
// FPToFixedJS()
// =============

// Converts a double precision floating point input value
// to a signed integer, with rounding to zero.

(bits(32), bit) FPToFixedJS(bits(64) op, FPCR_Type fpcr)

    // Unpack using fpcr to determine if subnormals are flushed-to-zero.
    (fptype,sign,value) = FPUnpack(op, fpcr);

    z = '1';
    // If NaN, set cumulative flag or take exception.
    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        FPProcessException(FPExc_InvalidOp, fpcr);
        z = '0';

    int_result = RoundDown(value);
    error = value - Real(int_result);

    // Determine whether supplied rounding mode requires an increment.

    round_it_up = (error != 0.0 && int_result < 0);
    if round_it_up then int_result = int_result + 1;

    integer result;
    if int_result < 0 then
        result = int_result - 2^32*RoundUp(Real(int_result)/Real(2^32));
    else
        result = int_result - 2^32*RoundDown(Real(int_result)/Real(2^32));

    // Generate exceptions.
    if int_result < -(2^31) || int_result > (2^31)-1 then
        FPProcessException(FPExc_InvalidOp, fpcr);
        z = '0';
    elsif error != 0.0 then
        FPProcessException(FPExc_Inexact, fpcr);
        z = '0';
    elsif sign == '1' && value == 0.0 then
        z = '0';
    elsif sign == '0' && value == 0.0 && !IsZero(op<51:0>) then
        z = '0';

    if fptype == FPType_Infinity then result = 0;

    return (result<31:0>, z);
```

G1.2.3.189 FPTwo

```
// FPTwo()
// =======

bits(N) FPTwo(bit sign, integer N)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 else (if N == 32 then 8 else 11));
    constant integer F = N - (E + 1);
    exp  = '1':Zeros(E-1);
    frac = Zeros(F);
    result = sign : exp : frac;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    return result;
```

G1.2.3.190 FPType

```
// FPType
// ======

enumeration FPType {FPType_Zero,
                    FPType_Denormal,
                    FPType_Nonzero,
                    FPType_Infinity,
                    FPType_QNaN,
                    FPType_SNaN};
```

G1.2.3.191 FPUnpack

```
// FPUnpack()
// ==========
//
// Used by data processing, int/fixed to FP and FP to int/fixed conversion instructions.
// For half-precision data it ignores AHP, and observes FZ16.

(FPType, bit, real) FPUnpack(bits(N) fpval, FPCR_Type fpcr_in)
    FPCR_Type fpcr = fpcr_in;
    fpcr.AHP = '0';
    (fp_type, sign, value) = FPUnpackBase(fpval, fpcr);
    return (fp_type, sign, value);
```

G1.2.3.192 FPUnpackBase

```
// FPUnpackBase()
// ==============
//
// Unpack a floating-point number into its type, sign bit and the real number
// that it represents. The real number result has the correct sign for numbers
// and infinities, is very large in magnitude for infinities, and is 0.0 for
// NaNs. (These values are chosen to simplify the description of comparisons
// and conversions.)
//
// The 'fpcr_in' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

(FPType, bit, real) FPUnpackBase(bits(N) fpval, FPCR_Type fpcr_in)
    assert N IN {16,32,64};

    constant FPCR_Type fpcr = fpcr_in;

    real value;
    bit sign;
    FPType fptype;

    if N == 16 then
        sign   = fpval<15>;
        exp16  = fpval<14:10>;
        frac16 = fpval<9:0>;
        if IsZero(exp16) then
            // Produce zero if value is zero or flush-to-zero is selected
            if IsZero(frac16) || fpcr.FZ16 == '1' then
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            fptype = FPType_Zero;  value = 0.0;
        else
            fptype = FPType_Denormal;  value = 2.0^-14 * (Real(UInt(frac16)) * 2.0^-10);
    elsif IsOnes(exp16) && fpcr.AHP == '0' then  // Infinity or NaN in IEEE format
        if IsZero(frac16) then
            fptype = FPType_Infinity;  value = 2.0^1000000;
        else
            fptype = if frac16<9> == '1' then FPType_QNaN else FPType_SNaN;
            value = 0.0;
    else
        fptype = FPType_Nonzero;
        value = 2.0^(UInt(exp16)-15) * (1.0 + Real(UInt(frac16)) * 2.0^-10);

elsif N == 32 then
    sign   = fpval<31>;
    exp32  = fpval<30:23>;
    frac32 = fpval<22:0>;

    if IsZero(exp32) then
        // Produce zero if value is zero or flush-to-zero is selected.
        if IsZero(frac32) || fpcr.FZ == '1' then
            fptype = FPType_Zero;  value = 0.0;
            if !IsZero(frac32) then  // Denormalized input flushed to zero
                FPProcessException(FPExc_InputDenorm, fpcr);
        else
            fptype = FPType_Denormal;  value = 2.0^-126 * (Real(UInt(frac32)) * 2.0^-23);
    elsif IsOnes(exp32) then
        if IsZero(frac32) then
            fptype = FPType_Infinity;  value = 2.0^1000000;
        else
            fptype = if frac32<22> == '1' then FPType_QNaN else FPType_SNaN;
            value = 0.0;
    else
        fptype = FPType_Nonzero;
        value = 2.0^(UInt(exp32)-127) * (1.0 + Real(UInt(frac32)) * 2.0^-23);

else // N == 64
    sign   = fpval<63>;
    exp64  = fpval<62:52>;
    frac64 = fpval<51:0>;

    if IsZero(exp64) then
        // Produce zero if value is zero or flush-to-zero is selected.
        if IsZero(frac64) || fpcr.FZ == '1' then
            fptype = FPType_Zero;  value = 0.0;
            if !IsZero(frac64) then  // Denormalized input flushed to zero
                FPProcessException(FPExc_InputDenorm, fpcr);
        else
            fptype = FPType_Denormal;  value = 2.0^-1022 * (Real(UInt(frac64)) * 2.0^-52);
    elsif IsOnes(exp64) then
        if IsZero(frac64) then
            fptype = FPType_Infinity;  value = 2.0^1000000;
        else
            fptype = if frac64<51> == '1' then FPType_QNaN else FPType_SNaN;
            value = 0.0;
    else
        fptype = FPType_Nonzero;
        value = 2.0^(UInt(exp64)-1023) * (1.0 + Real(UInt(frac64)) * 2.0^-52);

if sign == '1' then value = -value;

return (fptype, sign, value);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.3.193 FPUnpackCV

```
// FPUnpackCV()
// ============
//
// Used for FP to FP conversion instructions.
// For half-precision data ignores FZ16 and observes AHP.

(FPType, bit, real) FPUnpackCV(bits(N) fpval, FPCR_Type fpcr_in)
    FPCR_Type fpcr = fpcr_in;
    fpcr.FZ16 = '0';
    (fp_type, sign, value) = FPUnpackBase(fpval, fpcr);
    return (fp_type, sign, value);
```

G1.2.3.194 FPZero

```
// FPZero()
// ========

bits(N) FPZero(bit sign, integer N)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 else (if N == 32 then 8 else 11));
    constant integer F = N - (E + 1);
    exp  = Zeros(E);
    frac = Zeros(F);
    result = sign : exp : frac;
    return result;
```

G1.2.3.195 VFPExpandImm

```
// VFPExpandImm()
// ==============

bits(N) VFPExpandImm(bits(8) imm8, integer N)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 else (if N == 32 then 8 else 11));
    constant integer F = (N - E) - 1;
    sign = imm8<7>;
    exp  = NOT(imm8<6>):Replicate(imm8<6>, E-3):imm8<5:4>;
    frac = imm8<3:0>:Zeros(F-4);
    result = sign : exp : frac;

    return result;
```

G1.2.3.196 AddWithCarry

```
// AddWithCarry()
// ==============
// Integer addition with carry input, returning result and NZCV flags

(bits(N), bits(4)) AddWithCarry(bits(N) x, bits(N) y, bit carry_in)
    constant integer unsigned_sum = UInt(x) + UInt(y) + UInt(carry_in);
    constant integer signed_sum = SInt(x) + SInt(y) + UInt(carry_in);
    constant bits(N) result = unsigned_sum<N-1:0>; // same value as signed_sum<N-1:0>
    constant bit n = result<N-1>;
    constant bit z = if IsZero(result) then '1' else '0';
    constant bit c = if UInt(result) == unsigned_sum then '0' else '1';
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    constant bit v = if SInt(result) == signed_sum then '0' else '1';
    return (result, n:z:c:v);
```

G1.2.3.197 InterruptID

```
// InterruptID
// ===========

enumeration InterruptID {
    InterruptID_PMUIRQ,
    InterruptID_COMMIRQ,
    InterruptID_CTIIRQ,
    InterruptID_COMMRX,
    InterruptID_COMMTX,
    InterruptID_CNTP,
    InterruptID_CNTHP,
    InterruptID_CNTHPS,
    InterruptID_CNTPS,
    InterruptID_CNTV,
    InterruptID_CNTHV,
    InterruptID_CNTHVS,

};
```

G1.2.3.198 SetInterruptRequestLevel

```
// SetInterruptRequestLevel()
// ==========================
// Set a level-sensitive interrupt to the specified level.

SetInterruptRequestLevel(InterruptID id, Signal level);
```

G1.2.3.199 AArch64.BranchAddr

```
// AArch64.BranchAddr()
// ====================
// Return the virtual address with tag bits removed.
// This is typically used when the address will be stored to the program counter.

bits(64) AArch64.BranchAddr(bits(64) vaddress, bits(2) el)
    assert !UsingAArch32();
    constant integer msbit = AddrTop(vaddress, TRUE, el);
    if msbit == 63 then
        return vaddress;
    elsif el IN {EL0, EL1} && vaddress<msbit> == '1' then
        return SignExtend(vaddress<msbit:0>, 64);
    else
        return ZeroExtend(vaddress<msbit:0>, 64);
```

G1.2.3.200 AccessDescriptor

```
// AccessDescriptor
// ================
// Memory access or translation invocation details that steer architectural behavior
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
type AccessDescriptor is (
    AccessType acctype,
    bits(2) el,                     // Acting EL for the access
    SecurityState ss,               // Acting Security State for the access
    boolean acqsc,                  // Acquire with Sequential Consistency
    boolean acqpc,                  // FEAT_LRCPC: Acquire with Processor Consistency
    boolean relsc,                  // Release with Sequential Consistency
    boolean exclusive,              // Access has Exclusive semantics
    boolean atomicop,               // FEAT_LSE: Atomic read-modify-write access
    MemAtomicOp modop,              // FEAT_LSE: The modification operation in the 'atomicop' access
    boolean nontemporal,            // Hints the access is non-temporal
    boolean read,                   // Read from memory or only require read permissions
    boolean write,                  // Write to memory or only require write permissions
    CacheOp cacheop,                // DC/IC: Cache operation
    CacheOpScope opscope,           // DC/IC: Scope of cache operation
    CacheType cachetype,            // DC/IC: Type of target cache
    boolean pan,                    // FEAT_PAN: The access is subject to PSTATE.PAN
    boolean a32lsmd,                // A32 Load/Store Multiple Data access
    integer Rt,                     // Register named Rt in the instruction
    integer Rt2,                    // Register named Rt2 in the instruction
    integer Rs,                     // Register named Rs in the instruction
    integer Rs2,                    // Register named Rs2 in the instruction
    boolean ispair,                 // Access represents a Load/Store pair access
    boolean lowestaddress,          // Is the current access the lowest address accessed by
                                    // this instruction
)
```

G1.2.3.201 AccessType

```
// AccessType
// ==========

enumeration AccessType {
    AccessType_IFETCH,  // Instruction FETCH
    AccessType_GPR,     // Software load/store to a General Purpose Register
    AccessType_ASIMD,   // Software ASIMD extension load/store instructions
    AccessType_IC,      // Sysop IC
    AccessType_DC,      // Sysop DC (not DC {Z,G,GZ}VA)
    AccessType_DCZero,  // Sysop DC {Z,G,GZ}VA
    AccessType_AT,      // Sysop AT
    AccessType_TTW      // Translation Table Walk
};
```

G1.2.3.202 AddrTop

```
// AddrTop()
// =========
// Return the MSB number of a virtual address in the stage 1 translation regime for "el".
// If EL1 is using AArch64 then addresses from EL0 using AArch32 are zero-extended to 64 bits.

AddressSize AddrTop(bits(64) address, boolean IsInstr, bits(2) el)
    assert HaveEL(el);
    regime = S1TranslationRegime(el);
    if ELUsingAArch32(regime) then
        // AArch32 translation regime.
        return 31;
    else
        if EffectiveTBI(address, IsInstr, el) == '1' then
            return 55;
        else
            return 63;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.3.203 AddressSize

```
// AddressSize
// ===========

type AddressSize = integer;
```

G1.2.3.204 AlignmentEnforced

```
// AlignmentEnforced()
// ===================
// For the active translation regime, determine if alignment is required by all accesses

boolean AlignmentEnforced()
    constant Regime regime = TranslationRegime(PSTATE.EL);
    bit A;
    case regime of
        when Regime_EL2  A = SCTLR_EL2.A;
        when Regime_EL10 A = SCTLR_EL1.A;
        otherwise Unreachable();
    return A == '1';
```

G1.2.3.205 Allocation

```
// Allocation hints
// ================

constant bits(2) MemHint_No  = '00';    // No Read-Allocate, No Write-Allocate
constant bits(2) MemHint_WA  = '01';    // No Read-Allocate, Write-Allocate
constant bits(2) MemHint_RA  = '10';    // Read-Allocate, No Write-Allocate
constant bits(2) MemHint_RWA = '11';    // Read-Allocate, Write-Allocate
```

G1.2.3.206 BigEndian

```
// BigEndian()
// ===========

boolean BigEndian(AccessType acctype)
    boolean bigend;

    if UsingAArch32() then
        bigend = (PSTATE.E != '0');
    elsif PSTATE.EL == EL0 then
        bigend = (SCTLR_ELx[].E0E != '0');
    else
        bigend = (SCTLR_ELx[].EE != '0');
    return bigend;
```

G1.2.3.207 BigEndianReverse

```
// BigEndianReverse()
// ==================
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
bits(width) BigEndianReverse(bits(width) value)
    assert width IN {8, 16, 32, 64, 128, 256};

    if width == 8 then return value;
    return Reverse(value, 8);
```

G1.2.3.208 CacheOp

```
// CacheOp
// =======

enumeration CacheOp {
    CacheOp_Clean,
    CacheOp_Invalidate,
    CacheOp_CleanInvalidate
};
```

G1.2.3.209 CacheOpScope

```
// CacheOpScope
// ============

enumeration CacheOpScope {
    CacheOpScope_SetWay,
    CacheOpScope_PoU,
    CacheOpScope_PoC,
    CacheOpScope_PoP,
    CacheOpScope_PoDP,
    CacheOpScope_ALLU,
    CacheOpScope_ALLUIS
};
```

G1.2.3.210 CachePASpace

```
// CachePASpace
// ============

enumeration CachePASpace {
    CPAS_NonSecure,
    CPAS_SecureNonSecure,       // Applicable only for DC *SW / IC IALLU* in Secure state:
                                // match entries from Secure or Non-Secure PAS
    CPAS_Secure
};
```

G1.2.3.211 CacheType

```
// CacheType
// =========

enumeration CacheType {
    CacheType_Data,
    CacheType_Instruction
};
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.3.212 Cacheability

```
// Cacheability attributes
// =======================

constant bits(2) MemAttr_NC = '00';     // Non-cacheable
constant bits(2) MemAttr_WT = '10';     // Write-through
constant bits(2) MemAttr_WB = '11';     // Write-back
```

G1.2.3.213 CreateAccDescA32LSMD

```
// CreateAccDescA32LSMD()
// ======================
// Access descriptor for A32 loads/store multiple general purpose registers

AccessDescriptor CreateAccDescA32LSMD(MemOp memop)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GPR);

    accdesc.read            = memop == MemOp_LOAD;
    accdesc.write           = memop == MemOp_STORE;
    accdesc.pan             = TRUE;
    accdesc.a32lsmd         = TRUE;

    return accdesc;
```

G1.2.3.214 CreateAccDescASIMD

```
// CreateAccDescASIMD()
// ====================
// Access descriptor for ASIMD&FP loads/stores

AccessDescriptor CreateAccDescASIMD(MemOp memop, boolean nontemporal, boolean privileged)
    constant boolean ispair = FALSE;
    return CreateAccDescASIMD(memop, nontemporal, privileged, ispair);

// CreateAccDescASIMD()
// ====================

AccessDescriptor CreateAccDescASIMD(MemOp memop, boolean nontemporal, boolean privileged,
                                    boolean ispair)
    AccessDescriptor accdesc = NewAccDesc(AccessType_ASIMD);

    accdesc.nontemporal     = nontemporal;
    accdesc.el              = if !privileged then EL0 else PSTATE.EL;
    accdesc.read            = memop == MemOp_LOAD;
    accdesc.write           = memop == MemOp_STORE;
    accdesc.ispair          = ispair;
    accdesc.pan             = TRUE;
    return accdesc;
```

G1.2.3.215 CreateAccDescAT

```
// CreateAccDescAT()
// =================
// Access descriptor for address translation operations

AccessDescriptor CreateAccDescAT(SecurityState ss, bits(2) el, ATAccess ataccess)
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    AccessDescriptor accdesc = NewAccDesc(AccessType_AT);

    accdesc.el              = el;
    accdesc.ss              = ss;
    case ataccess of
        when ATAccess_Read
            (accdesc.read, accdesc.write, accdesc.pan) = (TRUE, FALSE, FALSE);
        when ATAccess_ReadPAN
            (accdesc.read, accdesc.write, accdesc.pan) = (TRUE, FALSE, TRUE);
        when ATAccess_Write
            (accdesc.read, accdesc.write, accdesc.pan) = (FALSE, TRUE, FALSE);
        when ATAccess_WritePAN
            (accdesc.read, accdesc.write, accdesc.pan) = (FALSE, TRUE, TRUE);

    return accdesc;
```

G1.2.3.216 CreateAccDescAcqRel

```
// CreateAccDescAcqRel()
// =====================
// Access descriptor for general purpose register loads/stores with ordering semantics

AccessDescriptor CreateAccDescAcqRel(MemOp memop, boolean acqsc)
    constant integer Rt = -1;
    return CreateAccDescAcqRel(memop, acqsc, Rt);

AccessDescriptor CreateAccDescAcqRel(MemOp memop, boolean acqsc, integer Rt)
    constant boolean ispair = FALSE;
    constant integer Rt2 = -1;
    return CreateAccDescAcqRel(memop, ispair, acqsc, Rt, Rt2);

AccessDescriptor CreateAccDescAcqRel(MemOp memop, boolean ispair, boolean acqsc, integer Rt,
                                     integer Rt2)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GPR);

    accdesc.acqsc           = acqsc;
    accdesc.relsc           = memop == MemOp_STORE;
    accdesc.read            = memop == MemOp_LOAD;
    accdesc.write           = memop == MemOp_STORE;
    accdesc.pan             = TRUE;
    accdesc.ispair          = ispair;
    accdesc.Rt              = Rt;
    accdesc.Rt2             = Rt2;

    return accdesc;
```

G1.2.3.217 CreateAccDescAtomicOp

```
// CreateAccDescAtomicOp()
// =======================
// Access descriptor for atomic read-modify-write memory accesses

AccessDescriptor CreateAccDescAtomicOp(MemAtomicOp modop, boolean acquire, boolean release,
                                       boolean privileged, integer Rt, integer Rs)
    constant integer Rt2 = -1;
    constant integer Rs2 = -1;

    return CreateAccDescAtomicOp(modop, acquire, release, privileged, Rt, Rt2, Rs, Rs2);

AccessDescriptor CreateAccDescAtomicOp(MemAtomicOp modop, boolean acquire, boolean release,
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                                    boolean privileged, integer Rt, integer Rt2, integer Rs,
                                    integer Rs2)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GPR);

    accdesc.acqsc           = acquire;
    accdesc.el              = if !privileged then EL0 else PSTATE.EL;
    accdesc.relsc           = release;
    accdesc.atomicop        = TRUE;
    accdesc.modop           = modop;
    accdesc.read            = TRUE;
    accdesc.write           = TRUE;
    accdesc.pan             = TRUE;
    accdesc.Rs              = Rs;
    accdesc.Rs2             = Rs2;
    accdesc.Rt              = Rt;
    accdesc.Rt2             = Rt2;

    return accdesc;
```

G1.2.3.218 CreateAccDescDC

```
// CreateAccDescDC()
// =================
// Access descriptor for data cache operations

AccessDescriptor CreateAccDescDC(CacheRecord cache)
    AccessDescriptor accdesc = NewAccDesc(AccessType_DC);

    accdesc.cacheop         = cache.cacheop;
    accdesc.cachetype       = cache.cachetype;
    accdesc.opscope         = cache.opscope;

    return accdesc;
```

G1.2.3.219 CreateAccDescDCZero

```
// CreateAccDescDCZero()
// =====================
// Access descriptor for data cache zero operations

AccessDescriptor CreateAccDescDCZero()
    AccessDescriptor accdesc = NewAccDesc(AccessType_DCZero);

    accdesc.write            = TRUE;
    accdesc.pan              = TRUE;

    return accdesc;
```

G1.2.3.220 CreateAccDescExLDST

```
// CreateAccDescExLDST()
// =====================
// Access descriptor for general purpose register loads/stores with exclusive semantics

AccessDescriptor CreateAccDescExLDST(MemOp memop, boolean acqrel, boolean privileged)
    constant integer Rt = -1;
    return CreateAccDescExLDST(memop, acqrel, privileged, Rt);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// CreateAccDescExLDST()
// =====================

AccessDescriptor CreateAccDescExLDST(MemOp memop, boolean acqrel, boolean privileged, integer Rt)
    constant boolean ispair = FALSE;
    constant integer Rt2 = -1;
    return CreateAccDescExLDST(memop, acqrel, privileged, ispair, Rt, Rt2);

// CreateAccDescExLDST()
// =====================

AccessDescriptor CreateAccDescExLDST(MemOp memop, boolean acqrel, boolean privileged,
                                     boolean ispair, integer Rt, integer Rt2)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GPR);

    accdesc.acqsc           = acqrel && memop == MemOp_LOAD;
    accdesc.relsc           = acqrel && memop == MemOp_STORE;
    accdesc.exclusive       = TRUE;
    accdesc.el              = if !privileged then EL0 else PSTATE.EL;
    accdesc.read            = memop == MemOp_LOAD;
    accdesc.write           = memop == MemOp_STORE;
    accdesc.ispair          = ispair;
    accdesc.pan             = TRUE;
    accdesc.Rt              = Rt;
    accdesc.Rt2             = Rt2;

    return accdesc;
```

G1.2.3.221 CreateAccDescGPR

```
// CreateAccDescGPR()
// ==================
// Access descriptor for general purpose register loads/stores
// without exclusive or ordering semantics

AccessDescriptor CreateAccDescGPR(MemOp memop, boolean nontemporal, boolean privileged)
    constant integer Rt = -1;
    return CreateAccDescGPR(memop, nontemporal, privileged, Rt);

AccessDescriptor CreateAccDescGPR(MemOp memop, boolean nontemporal, boolean privileged, integer Rt)
    constant boolean ispair = FALSE;
    constant integer Rt2 = -1;
    return CreateAccDescGPR(memop, nontemporal, privileged, ispair, Rt, Rt2);

AccessDescriptor CreateAccDescGPR(MemOp memop, boolean nontemporal, boolean privileged,
                                  boolean ispair, integer Rt, integer Rt2)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GPR);

    accdesc.el              = if !privileged then EL0 else PSTATE.EL;
    accdesc.nontemporal     = nontemporal;
    accdesc.read            = memop == MemOp_LOAD;
    accdesc.write           = memop == MemOp_STORE;
    accdesc.pan             = TRUE;
    accdesc.Rt              = Rt;
    accdesc.Rt2             = Rt2;
    accdesc.ispair          = ispair;

    return accdesc;
```

G1.2.3.222 CreateAccDescIC

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// CreateAccDescIC()
// =================
// Access descriptor for instruction cache operations

AccessDescriptor CreateAccDescIC(CacheRecord cache)
    AccessDescriptor accdesc = NewAccDesc(AccessType_IC);

    accdesc.cacheop         = cache.cacheop;
    accdesc.cachetype       = cache.cachetype;
    accdesc.opscope         = cache.opscope;

    return accdesc;
```

G1.2.3.223 CreateAccDesclFetch

```
// CreateAccDescIFetch()
// =====================
// Access descriptor for instruction fetches

AccessDescriptor CreateAccDescIFetch()
    constant AccessDescriptor accdesc = NewAccDesc(AccessType_IFETCH);

    return accdesc;
```

G1.2.3.224 CreateAccDescLDAcqPC

```
// CreateAccDescLDAcqPC()
// ======================
// Access descriptor for general purpose register loads with local ordering semantics

AccessDescriptor CreateAccDescLDAcqPC(boolean acquire, integer Rt)
    constant boolean ispair = FALSE;
    constant integer Rt2 = -1;
    return CreateAccDescLDAcqPC(ispair, acquire, Rt, Rt2);

AccessDescriptor CreateAccDescLDAcqPC(boolean ispair, boolean acquire, integer Rt, integer Rt2)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GPR);

    accdesc.acqpc           = acquire;
    accdesc.read            = TRUE;
    accdesc.pan             = TRUE;
    accdesc.ispair          = ispair;
    accdesc.Rt              = Rt;
    accdesc.Rt2             = Rt2;

    return accdesc;
```

G1.2.3.225 CreateAccDescS1TTW

```
// CreateAccDescS1TTW()
// ====================
// Access descriptor for stage 1 translation table walks

AccessDescriptor CreateAccDescS1TTW(AccessDescriptor accdesc_in)
    AccessDescriptor accdesc = NewAccDesc(AccessType_TTW);

    accdesc.el              = accdesc_in.el;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    accdesc.ss              = accdesc_in.ss;
    accdesc.read            = TRUE;

    return accdesc;
```

G1.2.3.226 CreateAccDescS2TTW

```
// CreateAccDescS2TTW()
// ====================
// Access descriptor for stage 2 translation table walks

AccessDescriptor CreateAccDescS2TTW(AccessDescriptor accdesc_in)
    AccessDescriptor accdesc = NewAccDesc(AccessType_TTW);

    accdesc.el              = accdesc_in.el;
    accdesc.ss              = accdesc_in.ss;
    accdesc.read            = TRUE;

    return accdesc;
```

G1.2.3.227 CreateAccDescTTEUpdate

```
// CreateAccDescTTEUpdate()
// ========================
// Access descriptor for translation table entry HW update

AccessDescriptor CreateAccDescTTEUpdate(AccessDescriptor accdesc_in)
    AccessDescriptor accdesc = NewAccDesc(AccessType_TTW);

    accdesc.el              = accdesc_in.el;
    accdesc.ss              = accdesc_in.ss;
    accdesc.atomicop        = TRUE;
    accdesc.modop           = MemAtomicOp_CAS;
    accdesc.read            = TRUE;
    accdesc.write           = TRUE;

    return accdesc;
```

G1.2.3.228 DataMemoryBarrier

```
// DataMemoryBarrier()
// ===================

DataMemoryBarrier(MBReqDomain domain, MBReqTypes types, boolean vmid_sensitive);
```

G1.2.3.229 DataSynchronizationBarrier

```
// DataSynchronizationBarrier()
// ============================

DataSynchronizationBarrier(MBReqDomain domain, MBReqTypes types, boolean vmid_sensitive);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.3.230 DeviceType

```
// DeviceType
// ==========
// Extended memory types for Device memory.

enumeration DeviceType {DeviceType_GRE, DeviceType_nGRE, DeviceType_nGnRE, DeviceType_nGnRnE};
```

G1.2.3.231 EffectiveTBI

```
// EffectiveTBI()
// ==============
// Returns the effective TBI in the AArch64 stage 1 translation regime for "el".

bit EffectiveTBI(bits(64) address, boolean IsInstr, bits(2) el)
    assert HaveEL(el);
    bit tbi;
    bit tbid;
    regime = TranslationRegime(el);

    case regime of
        when Regime_EL2
            tbi  = TCR_EL2.TBI;
            tbid = TCR_EL2.TBID;
        when Regime_EL10
            if AArch64.GetVARange(address) == VARange_LOWER then
                tbi  = TCR_EL1.TBI0;
                tbid = TCR_EL1.TBID0;
            else
                tbi  = TCR_EL1.TBI1;
                tbid = TCR_EL1.TBID1;

    return (if (tbi == '1' && (!IsFeatureImplemented(FEAT_PAuth) || tbid == '0' ||
            !IsInstr)) then '1' else '0');
```

G1.2.3.232 ErrorState

```
// ErrorState
// ==========
// The allowed error states that can be returned by memory and used by the PE.

enumeration ErrorState {ErrorState_UC,            // Uncontainable
                        ErrorState_UEU,           // Unrecoverable state
                        ErrorState_UEO,           // Restartable state
                        ErrorState_UER,           // Recoverable state
                        ErrorState_CE};          // Corrected
```

G1.2.3.233 Fault

```
// Fault
// =====
// Fault types.

enumeration Fault {Fault_None,
                   Fault_AccessFlag,
                   Fault_Alignment,
                   Fault_Background,
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                  Fault_Domain,
                  Fault_Permission,
                  Fault_Translation,
                  Fault_AddressSize,
                  Fault_SyncExternal,
                  Fault_SyncExternalOnWalk,
                  Fault_SyncParity,
                  Fault_SyncParityOnWalk,
                  Fault_AsyncParity,
                  Fault_AsyncExternal,
                  Fault_Debug,
                  Fault_TLBConflict,
                  Fault_HWUpdateAccessFlag,
                  Fault_Lockdown,
                  Fault_Exclusive,
                  Fault_ICacheMaint};
```

G1.2.3.234 FaultRecord

```
// FaultRecord
// ===========
// Fields that relate only to Faults.

type FaultRecord is (
    Fault            statuscode,        // Fault Status
    AccessDescriptor accessdesc,        // Details of the faulting access
    bits(64)         vaddress,          // Faulting virtual address
    FullAddress      ipaddress,         // Intermediate physical address
    boolean          s2fs1walk,         // Is on a Stage 1 translation table walk
    boolean          write,             // TRUE for a write, FALSE for a read
    integer          level,             // For translation, access flag and Permission faults
    bit              extflag,           // IMPLEMENTATION DEFINED syndrome for External aborts
    boolean          secondstage,       // Is a Stage 2 abort
    bits(4)          domain,            // Domain number, AArch32 only
    ErrorState       merrorstate,       // Incoming error state from memory
    WatchpointInfo   watchptinfo,       // Watchpoint related fields
    bits(4)          debugmoe           // Debug method of entry, from AArch32 only
)
```

G1.2.3.235 FullAddress

```
// FullAddress
// ===========
// Physical or Intermediate Physical Address type.
// Although AArch32 only has access to 40 bits of physical or intermediate physical address space,
// the full address type has 52 bits to allow interprocessing with AArch64.
// The maximum physical or intermediate physical address size is IMPLEMENTATION DEFINED,
// but never exceeds 52 bits.

type FullAddress is (
    PASpace  paspace,
    bits(52) address
)
```

G1.2.3.236 Hint_Prefetch

```
// Hint_Prefetch()
// ===============
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// Signals the memory system that memory accesses of type HINT to or from the specified address are
// likely in the near future. The memory system may take some action to speed up the memory
// accesses when they do occur, such as pre-loading the specified address into one or more
// caches as indicated by the innermost cache level target (0=L1, 1=L2, etc) and non-temporal hint
// stream. Any or all prefetch hints may be treated as a NOP. A prefetch hint must not cause a
// synchronous abort due to Alignment or Translation faults and the like. Its only effect on
// software-visible state should be on caches and TLBs associated with address, which must be
// accessible by reads, writes or execution, as defined in the translation regime of the current
// Exception level. It is guaranteed not to access Device memory.
// A Prefetch_EXEC hint must not result in an access that could not be performed by a speculative
// instruction fetch, therefore if all associated MMUs are disabled, then it cannot access any
// memory location that cannot be accessed by instruction fetches.

Hint_Prefetch(bits(64) address, PrefetchHint hint, integer target, boolean stream);
```

G1.2.3.237 IsWatchpointableAccess

```
// IsWatchpointableAccess()
// ========================
// Return TRUE if access should be checked for watchpoints.

boolean IsWatchpointableAccess(AccessDescriptor accdesc)
    return (!(accdesc.acctype IN {AccessType_IFETCH,
                                  AccessType_TTW,
                                  AccessType_DC,
                                  AccessType_IC,
                                  AccessType_AT}) ||
            (accdesc.acctype == AccessType_DC &&
             accdesc.cacheop == CacheOp_Invalidate &&
             (!UsingAArch32() || (boolean IMPLEMENTATION_DEFINED "DCIMVAC generates watchpoint"))));
```

G1.2.3.238 MBReqDomain

```
// MBReqDomain
// ===========
// Memory barrier domain.

enumeration MBReqDomain    {MBReqDomain_Nonshareable, MBReqDomain_InnerShareable,
                            MBReqDomain_OuterShareable, MBReqDomain_FullSystem};
```

G1.2.3.239 MBReqTypes

```
// MBReqTypes
// ==========
// Memory barrier read/write.

enumeration MBReqTypes     {MBReqTypes_Reads, MBReqTypes_Writes, MBReqTypes_All};
```

G1.2.3.240 MPURecord

```
// MPURecord
// =========

type MPURecord is (
    bit               CnP,             // [Armv8.2] TLB entry can be shared between different PEs
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    PASpace           paspace,
    Permissions       permissions,
    MemoryAttributes  memattrs
)
```

G1.2.3.241 MemAtomicOp

```
// MemAtomicOp
// ===========
// Atomic data processing instruction types.

enumeration MemAtomicOp {
    MemAtomicOp_ADD,
    MemAtomicOp_BIC,
    MemAtomicOp_EOR,
    MemAtomicOp_ORR,
    MemAtomicOp_SMAX,
    MemAtomicOp_SMIN,
    MemAtomicOp_UMAX,
    MemAtomicOp_UMIN,
    MemAtomicOp_SWP,
    MemAtomicOp_CAS,
};
```

G1.2.3.242 MemAttrHints

```
// MemAttrHints
// ============
// Attributes and hints for Normal memory.

type MemAttrHints is (
    bits(2) attrs,  // See MemAttr_*, Cacheability attributes
    bits(2) hints,  // See MemHint_*, Allocation hints
    boolean transient
)
```

G1.2.3.243 MemOp

```
// MemOp
// =====
// Memory access instruction types.

enumeration MemOp {MemOp_LOAD, MemOp_STORE, MemOp_PREFETCH};
```

G1.2.3.244 MemType

```
// MemType
// =======
// Basic memory types.

enumeration MemType {MemType_Normal, MemType_Device};
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.3.245 Memory

```
// Memory Tag type
// ===============

enumeration MemTagType {
    MemTag_AllocationTagged,
    MemTag_Untagged
};
```

G1.2.3.246 MemoryAttributes

```
// MemoryAttributes
// ================
// Memory attributes descriptor

type MemoryAttributes is (
    MemType      memtype,
    DeviceType   device,       // For Device memory types
    MemAttrHints inner,        // Inner hints and attributes
    MemAttrHints outer,        // Outer hints and attributes
    Shareability shareability, // Shareability attribute
)
```

G1.2.3.247 NewAccDesc

```
// NewAccDesc()
// ============
// Create a new AccessDescriptor with initialised fields

AccessDescriptor NewAccDesc(AccessType acctype)
    AccessDescriptor accdesc;

    accdesc.acctype             = acctype;
    accdesc.el                  = PSTATE.EL;
    accdesc.ss                  = SecurityStateAtEL(PSTATE.EL);
    accdesc.acqsc               = FALSE;
    accdesc.acqpc               = FALSE;
    accdesc.relsc               = FALSE;
    accdesc.exclusive           = FALSE;
    accdesc.atomicop            = FALSE;
    accdesc.nontemporal         = FALSE;
    accdesc.read                = FALSE;
    accdesc.write               = FALSE;
    accdesc.pan                 = FALSE;
    accdesc.a32lsmd             = FALSE;
    accdesc.Rs                  = -1;
    accdesc.Rs2                 = -1;
    accdesc.Rt                  = -1;
    accdesc.Rt2                 = -1;
    accdesc.ispair              = FALSE;
    accdesc.lowestaddress       = TRUE;

    return accdesc;
```

G1.2.3.248 PASpace

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// PASpace
// =======
// Physical address spaces

enumeration PASpace {
    PAS_Secure,
    PAS_NonSecure
};
```

G1.2.3.249 Permissions

```
// Permissions
// ===========
// Access Control bits in translation table descriptors

type Permissions is (
    bits(2) ap_table,   // Stage 1 hierarchical access permissions
    bit     xn_table,   // Stage 1 hierarchical execute-never for single EL regimes
    bit     pxn_table,  // Stage 1 hierarchical privileged execute-never
    bit     uxn_table,  // Stage 1 hierarchical unprivileged execute-never
    bits(3) ap,         // Stage 1 access permissions
    bit     xn,         // Stage 1 execute-never for single EL regimes
    bit     uxn,        // Stage 1 unprivileged execute-never
    bit     pxn,        // Stage 1 privileged execute-never
    bits(2) s2ap,       // Stage 2 access permissions
    bit     s2xnx,      // Stage 2 extended execute-never
    bit     dbm,        // Dirty bit management
    bit     s2xn        // Stage 2 execute-never
)
```

G1.2.3.250 PhysMemRead

```
// PhysMemRead()
// =============
// Returns the value read from memory, and a status.
// Returned value is UNKNOWN if an External abort occurred while reading the
// memory.
// Otherwise the PhysMemRetStatus statuscode is Fault_None.

(PhysMemRetStatus, bits(8*size)) PhysMemRead(AddressDescriptor desc, integer size,
                                             AccessDescriptor accdesc);
```

G1.2.3.251 PhysMemRetStatus

```
// PhysMemRetStatus
// ================
// Fields that relate only to return values of PhysMem functions.

type PhysMemRetStatus is (
    Fault       statuscode,     // Fault Status
    bit         extflag,        // IMPLEMENTATION DEFINED syndrome for External aborts
    ErrorState  merrorstate,    // Optional error state returned on a physical memory access
)
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.3.252 PhysMemWrite

```
// PhysMemWrite()
// ==============
// Writes the value to memory, and returns the status of the write.
// If there is an External abort on the write, the PhysMemRetStatus indicates this.
// Otherwise the statuscode of PhysMemRetStatus is Fault_None.

PhysMemRetStatus PhysMemWrite(AddressDescriptor desc, integer size, AccessDescriptor accdesc,
                              bits(8*size) value);
```

G1.2.3.253 PrefetchHint

```
// PrefetchHint
// ============
// Prefetch hint types.

enumeration PrefetchHint {Prefetch_READ, Prefetch_WRITE, Prefetch_EXEC};
```

G1.2.3.254 S1AccessControls

```
// S1AccessControls
// ================
// Effective access controls defined by stage 1 translation

type S1AccessControls is (
    bit r,                // Stage 1 base read permission
    bit w,                // Stage 1 base write permission
    bit x,                // Stage 1 base execute permission
    bit wxn               // Stage 1 write permission implies execute-never

)
```

G1.2.3.255 S2AccessControls

```
// S2AccessControls
// ================
// Effective access controls defined by stage 2 translation

type S2AccessControls is (
    bit r,                // Stage 2 read permission.
    bit w,                // Stage 2 write permission.
    bit x,                // Stage 2 execute permission.
)
```

G1.2.3.256 Shareability

```
// Shareability
// ============

enumeration Shareability {
    Shareability_NSH,
    Shareability_ISH,
    Shareability_OSH
};
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.3.257 SpeculativeStoreBypassBarrierToPA

```
// SpeculativeStoreBypassBarrierToPA()
// ===================================

SpeculativeStoreBypassBarrierToPA();
```

G1.2.3.258 SpeculativeStoreBypassBarrierToVA

```
// SpeculativeStoreBypassBarrierToVA()
// ===================================

SpeculativeStoreBypassBarrierToVA();
```

G1.2.3.259 VARange

```
// VARange
// =======
// Virtual address ranges

enumeration VARange {
    VARange_LOWER,
    VARange_UPPER
};
```

G1.2.3.260 ASID

```
// ASID
// ====
// Effective ASID.

bits(16) ASID[]
    if TCR_EL1.A1 == '1' then
        return TTBR1_EL1.ASID;
    else
        return TTBR0_EL1.ASID;
```

G1.2.3.261 ExecutionCntxt

```
// ExecutionCntxt
// ==============
// Context information for prediction restriction operation.

type ExecutionCntxt is (
    boolean          is_vmid_valid, // is vmid valid for current context
    boolean          all_vmid,      // should the operation be applied for all vmids
    bits(16)         vmid,          // if all_vmid = FALSE, vmid to which operation is applied
    boolean          is_asid_valid, // is asid valid for current context
    boolean          all_asid,      // should the operation be applied for all asids
    bits(16)         asid,          // if all_asid = FALSE, ASID to which operation is applied
    bits(2)          target_el,     // target EL at which operation is performed
    SecurityState    security,
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    RestrictType     restriction    // type of restriction operation
)
```

G1.2.3.262 RESTRICT_PREDICTIONS

```
// RESTRICT_PREDICTIONS()
// ======================
// Clear all speculated values.

RESTRICT_PREDICTIONS(ExecutionCntxt c)
    IMPLEMENTATION_DEFINED;
```

G1.2.3.263 RestrictType

```
// RestrictType
// ============
// Type of restriction on speculation.

enumeration RestrictType {
    RestrictType_DataValue,
    RestrictType_ControlFlow,
    RestrictType_CachePrefetch,
};
```

G1.2.3.264 TargetSecurityState

```
// TargetSecurityState()
// =====================
// Decode the target security state for the prediction context.

SecurityState TargetSecurityState(bit NS)
    curr_ss =  SecurityStateAtEL(PSTATE.EL);
    if curr_ss == SS_NonSecure then
        return SS_NonSecure;
    elsif curr_ss == SS_Secure then
        case NS of
            when '0' return SS_Secure;
            when '1' return SS_NonSecure;
    Unreachable();
```

G1.2.3.265 BranchTo

```
// BranchTo()
// ==========
// Set program counter to a new address, with a branch type.
// Parameter branch_conditional indicates whether the executed branch has a conditional encoding.
// In AArch64 state the address might include a tag in the top eight bits.

BranchTo(bits(N) target, BranchType branch_type, boolean branch_conditional)
    Hint_Branch(branch_type);
    if N == 32 then
        assert UsingAArch32();
        _PC = ZeroExtend(target, 64);
    else
        assert N == 64 && !UsingAArch32();
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        constant bits(64) target_vaddress = AArch64.BranchAddr(target<63:0>, PSTATE.EL);

        _PC = target_vaddress;
    return;
```

G1.2.3.266 BranchToAddr

```
// BranchToAddr()
// ==============
// Set program counter to a new address, with a branch type.
// In AArch64 state the address does not include a tag in the top eight bits.

BranchToAddr(bits(N) target, BranchType branch_type)
    Hint_Branch(branch_type);
    if N == 32 then
        assert UsingAArch32();
        _PC = ZeroExtend(target, 64);
    else
        assert N == 64 && !UsingAArch32();
        _PC = target<63:0>;
    return;
```

G1.2.3.267 BranchType

```
// BranchType
// ==========
// Information associated with a change in control flow.

enumeration BranchType {
    BranchType_DIRCALL,     // Direct Branch with link
    BranchType_INDCALL,     // Indirect Branch with link
    BranchType_ERET,        // Exception return (indirect)
    BranchType_DBGEXIT,     // Exit from Debug state
    BranchType_RET,         // Indirect branch with function return hint
    BranchType_DIR,         // Direct branch
    BranchType_INDIR,       // Indirect branch
    BranchType_EXCEPTION,   // Exception entry
    BranchType_RESET,       // Reset
    BranchType_UNKNOWN};   // Other
```

G1.2.3.268 ESize

```
// SIMD and Floating-point registers
// ++++++++++++++++++++++++++++++++

// ESize
// =====

type ESize = integer;
```

G1.2.3.269 EffectiveFPCR

```
// EffectiveFPCR()
// ===============
// Returns the effective FPCR value
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
FPCR_Type EffectiveFPCR()
    return FPCR;
```

G1.2.3.270 FPCR_Type

```
// FPCR_Type
// =========
// A type representing the FPCR register

type FPCR_Type;
```

G1.2.3.271 FPMR_Type

```
// FPMR_Type
// =========
// A type representing the FPMR register

type FPMR_Type;
```

G1.2.3.272 Hint_Branch

```
// Hint_Branch()
// =============
// Report the hint passed to BranchTo() and BranchToAddr(), for consideration when processing
// the next instruction.

Hint_Branch(BranchType hint);
```

G1.2.3.273 NextInstrAddr

```
// NextInstrAddr()
// ===============
// Return address of the sequentially next instruction.

bits(N) NextInstrAddr(integer N);
```

G1.2.3.274 ResetExternalDebugRegisters

```
// ResetExternalDebugRegisters()
// =============================
// Reset the External Debug registers in the Core power domain.

ResetExternalDebugRegisters(boolean cold_reset);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.3.275 ThisInstrAddr

```
// ThisInstrAddr()
// ===============
// Return address of the current instruction.

bits(N) ThisInstrAddr(integer N)
    assert N == 64 || (N == 32 && UsingAArch32());
    return _PC<N-1:0>;
```

G1.2.3.276 UnimplementedIDRegister

```
// UnimplementedIDRegister()
// =========================
// Trap access to unimplemented encodings in the feature ID register space.

UnimplementedIDRegister()
    if IsFeatureImplemented(FEAT_IDST) then
        target_el = PSTATE.EL;
        if PSTATE.EL == EL0 then
            target_el = if EL2Enabled() && HCR_EL2.TGE == '1' then EL2 else EL1;
        AArch64.SystemAccessTrap(target_el, 0x18);
    UNDEFINED;
```

G1.2.3.277 V

```
// V - setter
// ==========
// Write to SIMD&FP register with implicit extension from
// 8, 16, 32, 64 or 128 bits.

V[integer n, ESize width] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width IN {8, 16, 32, 64, 128};
    _V[n] = ZeroExtend(value, 128);
    return;

// V - getter
// ==========
// Read from SIMD&FP register with implicit slice of 8, 16
// 32, 64 or 128 bits.

bits(width) V[integer n, ESize width]
    assert n >= 0 && n <= 31;
    assert width IN {8, 16, 32, 64, 128};
    return _V[n]<width-1:0>;
```

G1.2.3.278 Vpart

```
// Vpart - getter
// ==============
// Reads a 128-bit SIMD&FP register in up to two parts:
//  part 0 returns the bottom 8, 16, 32 or 64 bits of a value held in the register;
//  part 1 returns the top half of the bottom 64 bits or the top half of the 128-bit
//  value held in the register.

bits(width) Vpart[integer n, integer part, ESize width]
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    assert n >= 0 && n <= 31;
    assert part IN {0, 1};
    if part == 0 then
        assert width IN {8,16,32,64};
        return _V[n]<width-1:0>;
    else
        assert width IN {32,64};
        return _V[n]<(width * 2)-1:width>;

// Vpart - setter
// ==============
// Writes a 128-bit SIMD&FP register in up to two parts:
//  part 0 zero extends a 8, 16, 32, or 64-bit value to fill the whole register;
//  part 1 inserts a 64-bit value into the top half of the register.

Vpart[integer n, integer part, ESize width] = bits(width) value
    assert n >= 0 && n <= 31;
    assert part IN {0, 1};
    if part == 0 then
        assert width IN {8,16,32,64};
        _V[n] = ZeroExtend(value, 128);
    else
        assert width == 64;
        _V[n]<(width * 2)-1:width> = value<width-1:0>;
```

G1.2.3.279 _PC

```
// _PC - the program counter
// =========================

bits(64) _PC;
```

G1.2.3.280 _R

```
// _R - the general-purpose register file array
// ============================================

array bits(64) _R[0..30];
```

G1.2.3.281 _V

```
// _V - the SIMD&FP register file array
// ====================================

array bits(128) _V[0..31];
```

G1.2.3.282 SPSR_ELx

```
// SPSR_ELx - getter
// =================

bits(64) SPSR_ELx[]
    bits(64) result;
    case PSTATE.EL of
        when EL1          result = SPSR_EL1<63:0>;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when EL2          result = SPSR_EL2<63:0>;
        otherwise         Unreachable();
    return result;

// SPSR_ELx - setter
// =================

SPSR_ELx[] = bits(64) value
    case PSTATE.EL of
        when EL1          SPSR_EL1<63:0> = value<63:0>;
        when EL2          SPSR_EL2<63:0> = value<63:0>;
        otherwise         Unreachable();
    return;
```

G1.2.3.283 AddressAdd

```
// AddressAdd()
// ============
// Add an address with an offset and return the result.

bits(64) AddressAdd(bits(64) base, integer offset, AccessDescriptor accdesc)
    return AddressAdd(base, offset<63:0>, accdesc);

bits(64) AddressAdd(bits(64) base, bits(64) offset, AccessDescriptor accdesc)
    bits(64) result = base + offset;
    return result;
```

G1.2.3.284 AddressInNaturallyAlignedBlock

```
// AddressInNaturallyAlignedBlock()
// ================================
// Returns TRUE if the given addresses are in the same naturally aligned block, and FALSE otherwise.
// An address is in a naturally aligned block if it meets any of the below conditions:
// * is a power-of-two size.
// * Is no larger than the DC ZVA block size if ESR_ELx.FnP is being set to 0b0, or EDHSR is not
//   implemented or EDHSR.FnP is being set to 0b0 (as appropriate).
// * Is no larger than the smallest implemented translation granule if ESR_ELx.FnP, or EDHSR.FnP
//   (as appropriate) is being set to 0b1.
// * Contains a watchpointed address accessed by the memory access or set of contiguous memory
//   accesses that triggered the watchpoint.

boolean AddressInNaturallyAlignedBlock(bits(64) address1, bits(64) address2);
```

G1.2.3.285 AddressIncrement

```
// AddressIncrement()
// ==================
// Increment an address and return the result.

bits(64) AddressIncrement(bits(64) base, integer increment, AccessDescriptor accdesc)
    return AddressIncrement(base, increment<63:0>, accdesc);

bits(64) AddressIncrement(bits(64) base, bits(64) increment, AccessDescriptor accdesc)
    bits(64) result = base + increment;
    return result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.3.286 ClearEventRegister

```
// ClearEventRegister()
// ====================
// Clear the Event Register of this PE.

ClearEventRegister()
    EventRegister = '0';
    return;
```

G1.2.3.287 ConditionHolds

```
// ConditionHolds()
// ================
// Return TRUE iff COND currently holds

boolean ConditionHolds(bits(4) cond)
    // Evaluate base condition.
    boolean result;
    case cond<3:1> of
        when '000' result = (PSTATE.Z == '1');                          // EQ or NE
        when '001' result = (PSTATE.C == '1');                          // CS or CC
        when '010' result = (PSTATE.N == '1');                          // MI or PL
        when '011' result = (PSTATE.V == '1');                          // VS or VC
        when '100' result = (PSTATE.C == '1' && PSTATE.Z == '0');       // HI or LS
        when '101' result = (PSTATE.N == PSTATE.V);                     // GE or LT
        when '110' result = (PSTATE.N == PSTATE.V && PSTATE.Z == '0');  // GT or LE
        when '111' result = TRUE;                                       // AL

    // Condition flag values in the set '111x' indicate always true
    // Otherwise, invert condition if necessary.
    if cond<0> == '1' && cond != '1111' then
        result = !result;

    return result;
```

G1.2.3.288 ConsumptionOfSpeculativeDataBarrier

```
// ConsumptionOfSpeculativeDataBarrier()
// =====================================

ConsumptionOfSpeculativeDataBarrier();
```

G1.2.3.289 CurrentInstrSet

```
// CurrentInstrSet()
// =================

InstrSet CurrentInstrSet()
    return InstrSet_A64;
```

G1.2.3.290 CurrentSecurityState

```
// CurrentSecurityState()
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// ======================
// Returns the effective security state at the exception level based off current settings.

SecurityState CurrentSecurityState()
    // Always in Secure state
    return SS_Secure;
```

G1.2.3.291 EL0

```
// EL0-3
// =====
// PSTATE.EL Exception level bits.

constant bits(2) EL3 = '11';
constant bits(2) EL2 = '10';
constant bits(2) EL1 = '01';
constant bits(2) EL0 = '00';
```

G1.2.3.292 EL2Enabled

```
// EL2Enabled()
// ============
// Returns TRUE if EL2 is present and executing
// - with the PE in Non-secure state when Non-secure EL2 is implemented, or
// - with the PE in Secure state when Secure EL2 is implemented and enabled, or
// - when EL3 is not implemented.

boolean EL2Enabled()
    return TRUE;
```

G1.2.3.293 ELFromSPSR

```
// ELFromSPSR()
// ============

// Convert an SPSR value encoding to an Exception level.
// Returns (valid,EL):
//   'valid' is TRUE if 'spsr<4:0>' encodes a valid mode for the current state.
//   'EL'    is the Exception level decoded from 'spsr'.

(boolean,bits(2)) ELFromSPSR(bits(N) spsr)
    bits(2) el;
    boolean valid;
    if spsr<4> == '0' then      // AArch64 state
        el = spsr<3:2>;
        if !HaveAArch64() then
            valid = FALSE;      // No AArch64 support
        elsif !HaveEL(el) then
            valid = FALSE;      // Exception level not implemented
        elsif spsr<1> == '1' then
            valid = FALSE;      // M<1> must be 0
        elsif el == EL0 && spsr<0> == '1' then
            valid = FALSE;      // for EL0, M<0> must be 0
        else
            valid = TRUE;
    else
        valid = FALSE;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if !valid then el = bits(2) UNKNOWN;
    return (valid,el);
```

G1.2.3.294 ELUsingAArch32

```
// ELUsingAArch32()
// ================

boolean ELUsingAArch32(bits(2) el)
    return FALSE;
```

G1.2.3.295 ELUsingAArch32K

```
// ELUsingAArch32K()
// =================

(boolean,boolean) ELUsingAArch32K(bits(2) el)
    return (TRUE, FALSE);
```

G1.2.3.296 EffectiveEA

```
// EffectiveEA()
// =============
// Returns effective SCR_EL3.EA value

bit EffectiveEA()
    return '0';
```

G1.2.3.297 EffectiveHCR_EL2_NVx

```
// EffectiveHCR_EL2_NVx()
// ======================
// Return the Effective value of HCR_EL2.<NV2,NV1,NV>.

bits(3) EffectiveHCR_EL2_NVx()
    return '000';
```

G1.2.3.298 EffectiveSCR_EL3_NS

```
// EffectiveSCR_EL3_NS()
// =====================
// Return Effective SCR_EL3.NS value.

bit EffectiveSCR_EL3_NS()
    return if SecureOnlyImplementation() then '0' else '1';
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.3.299 EffectiveSCR_EL3_RW

```
// EffectiveSCR_EL3_RW()
// =====================
// Returns effective SCR_EL3.RW value

bit EffectiveSCR_EL3_RW()
    return '1';
```

G1.2.3.300 EffectiveTGE

```
// EffectiveTGE()
// ==============
// Returns effective TGE value

bit EffectiveTGE()
    return if EL2Enabled() then HCR_EL2.TGE else '0';
```

G1.2.3.301 EndOfInstruction

```
// EndOfInstruction()
// ==================
// Terminate processing of the current instruction.

EndOfInstruction();
```

G1.2.3.302 EnterLowPowerState

```
// EnterLowPowerState()
// ====================
// PE enters a low-power state.

EnterLowPowerState();
```

G1.2.3.303 EventRegister

```
// EventRegister
// =============
// Event Register for this PE.

bits(1) EventRegister;
```

G1.2.3.304 ExceptionalOccurrenceTargetState

```
// ExceptionalOccurrenceTargetState
// ================================
// Enumeration to represent the target state of an Exceptional Occurrence.
// The Exceptional Occurrence can be either Exception or Debug State entry.

enumeration ExceptionalOccurrenceTargetState {
    AArch32_NonDebugState,
    AArch64_NonDebugState,
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    DebugState
};
```

G1.2.3.305 ExecuteAsNOP

```
// ExecuteAsNOP()
// ==============

ExecuteAsNOP()
    EndOfInstruction();
```

G1.2.3.306 FIQPending

```
// FIQPending()
// ============
// Returns TRUE if there is any pending physical FIQ.

boolean FIQPending();
```

G1.2.3.307 GetAccumulatedFPExceptions

```
// GetAccumulatedFPExceptions()
// ============================
// Returns FP exceptions accumulated by the PE.

bits(8) GetAccumulatedFPExceptions();
```

G1.2.3.308 GetPSRFromPSTATE

```
// GetPSRFromPSTATE()
// ==================
// Return a PSR value which represents the current PSTATE

bits(N) GetPSRFromPSTATE(ExceptionalOccurrenceTargetState targetELState, integer N)
    if UsingAArch32() && (targetELState IN {AArch32_NonDebugState, DebugState}) then
        assert N == 32;
    else
        assert N == 64;

    bits(N) spsr = Zeros(N);
    spsr<31:28> = PSTATE.<N,Z,C,V>;
    if IsFeatureImplemented(FEAT_PAN) then spsr<22> = PSTATE.PAN;
    spsr<20>     = PSTATE.IL;
    if IsFeatureImplemented(FEAT_DIT) then spsr<24> = PSTATE.DIT;
    if IsFeatureImplemented(FEAT_UAO) then spsr<23> = PSTATE.UAO;
    spsr<21>    = PSTATE.SS;
    if IsFeatureImplemented(FEAT_SSBS) then spsr<12> = PSTATE.SSBS;
    spsr<9:6>   = PSTATE.<D,A,I,F>;
    spsr<4>     = PSTATE.nRW;
    spsr<3:2>   = PSTATE.EL;
    spsr<0>     = PSTATE.SP;
    return spsr;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.3.309 HaveAArch32EL

```
// HaveAArch32EL()
// ===============
// Return TRUE if Exception level 'el' supports AArch32 in this implementation

boolean HaveAArch32EL(bits(2) el)
    return FALSE;
```

G1.2.3.310 HaveAArch64

```
// HaveAArch64()
// =============
// Return TRUE if the highest Exception level is using AArch64 state.

boolean HaveAArch64()
    return (IsFeatureImplemented(FEAT_AA64EL0)
            || IsFeatureImplemented(FEAT_AA64EL1)
            || IsFeatureImplemented(FEAT_AA64EL2)
            );
```

G1.2.3.311 HaveEL

```
// HaveEL()
// ========
// Return TRUE if Exception level 'el' is supported

boolean HaveEL(bits(2) el)
    case el of
        when EL1,EL0
            return TRUE;                        // EL1 and EL0 must exist
        when EL2
            return IsFeatureImplemented(FEAT_AA64EL2);
        when EL3
            return FALSE;
        otherwise
            Unreachable();
```

G1.2.3.312 HaveELUsingSecurityState

```
// HaveELUsingSecurityState()
// ==========================
// Returns TRUE if Exception level 'el' with Security state 'secure' is supported,
// FALSE otherwise.

boolean HaveELUsingSecurityState(bits(2) el, boolean secure)

    case el of
        when EL3
            assert secure;
            return HaveEL(EL3);
        when EL2
            if secure then
                return HaveEL(EL2) && IsFeatureImplemented(FEAT_SEL2);
            else
                return HaveEL(EL2);
        otherwise
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            return (HaveEL(EL3) ||
                    (secure == boolean IMPLEMENTATION_DEFINED "Secure-only implementation"));
```

G1.2.3.313 HaveSecureState

```
// HaveSecureState()
// =================
// Return TRUE if Secure State is supported.

boolean HaveSecureState()
    return SecureOnlyImplementation();
```

G1.2.3.314 HighestEL

```
// HighestEL()
// ===========
// Returns the highest implemented Exception level.

bits(2) HighestEL()
    if HaveEL(EL3) then
        return EL3;
    elsif HaveEL(EL2) then
        return EL2;
    else
        return EL1;
```

G1.2.3.315 Hint_DGH

```
// Hint_DGH()
// ==========
// Provides a hint to close any gathering occurring within the implementation.

Hint_DGH();
```

G1.2.3.316 Hint_WFE

```
// Hint_WFE()
// ==========
// Provides a hint indicating that the PE can enter a low-power state and
// remain there until a wakeup event occurs.

Hint_WFE()
    if IsEventRegisterSet() then
        ClearEventRegister();
        return;

    boolean trap;
    bits(2) target_el;
    (trap, target_el) = AArch64.CheckForWFxTrap(WFxType_WFE);
    if trap then
        AArch64.WFxTrap(WFxType_WFE, target_el);
    else
        WaitForEvent();
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.3.317 Hint_WFI

```
// Hint_WFI()
// ==========
// Provides a hint indicating that the PE can enter a low-power state and
// remain there until a wakeup event occurs.

Hint_WFI()
    if AArch64.InterruptPending() then
        // No further operation if an interrupt is pending.
        EndOfInstruction();
        return;

    boolean trap;
    bits(2) target_el;
    (trap, target_el) = AArch64.CheckForWFxTrap(WFxType_WFI);
    if trap then
        AArch64.WFxTrap(WFxType_WFI, target_el);
    else
        WaitForInterrupt();
```

G1.2.3.318 Hint_Yield

```
// Hint_Yield()
// ============
// Provides a hint that the task performed by a thread is of low
// importance so that it could yield to improve overall performance.

Hint_Yield();
```

G1.2.3.319 IRQPending

```
// IRQPending()
// ============
// Returns TRUE if there is any pending physical IRQ.

boolean IRQPending();
```

G1.2.3.320 IllegalExceptionReturn

```
// IllegalExceptionReturn()
// ========================

boolean IllegalExceptionReturn(bits(N) spsr)
    // Check for illegal return:
    //   * To an unimplemented Exception level.
    //   * To EL2 in Secure state, when SecureEL2 is not enabled.
    //   * To EL0 using AArch64 state, with SPSR.M<0>==1.
    //   * To AArch64 state with SPSR.M<1>==1.
    //   * To AArch32 state with an illegal value of SPSR.M.
    (valid, target) = ELFromSPSR(spsr);
    if !valid then return TRUE;

    // Check for return to higher Exception level.
    if UInt(target) > UInt(PSTATE.EL) then return TRUE;

    spsr_mode_is_aarch32 = (spsr<4> == '1');
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    // Check for illegal return:
    //   * To EL1, EL2 or EL3 with register width specified in the SPSR different from the
    //     Execution state used in the Exception level being returned to, as determined by
    //     the SCR_EL3.RW or HCR_EL2.RW bits, or as configured from reset.
    //   * To EL0 using AArch64 state when EL1 is using AArch32 state as determined by the
    //     SCR_EL3.RW or HCR_EL2.RW bits or as configured from reset.
    //   * To AArch64 state from AArch32 state (should be caught by above).
    (known, target_el_is_aarch32) = ELUsingAArch32K(target);
    assert known || (target == EL0 && !ELUsingAArch32(EL1));
    if known && spsr_mode_is_aarch32 != target_el_is_aarch32 then return TRUE;

    // Check for illegal return from AArch32 to AArch64.
    if UsingAArch32() && !spsr_mode_is_aarch32 then return TRUE;
    // Check for illegal return to EL1 when HCR_EL2.TGE is set and when either of
    // * SecureEL2 is enabled.
    // * SecureEL2 is not enabled and EL1 is in Non-secure state.
    if EL2Enabled() && target == EL1 && HCR_EL2.TGE == '1' then
        if (!IsSecureBelowEL3() || IsSecureEL2Enabled()) then return TRUE;

    return FALSE;
```

G1.2.3.321 InstrSet

```
// InstrSet
// ========

enumeration InstrSet {InstrSet_A64, InstrSet_A32, InstrSet_T32};
```

G1.2.3.322 InstructionFetchBarrier

```
// InstructionFetchBarrier()
// =========================

InstructionFetchBarrier();
```

G1.2.3.323 InstructionSynchronizationBarrier

```
// InstructionSynchronizationBarrier()
// ===================================

InstructionSynchronizationBarrier();
```

G1.2.3.324 IsASEInstruction

```
// IsASEInstruction()
// ==================
// Returns TRUE if the current instruction is an ASIMD or SVE vector instruction.

boolean IsASEInstruction();
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.3.325 IsCurrentSecurityState

```
// IsCurrentSecurityState()
// ========================
// Returns TRUE if the current Security state matches
// the given Security state, and FALSE otherwise.

boolean IsCurrentSecurityState(SecurityState ss)
    return CurrentSecurityState() == ss;
```

G1.2.3.326 IsEventRegisterSet

```
// IsEventRegisterSet()
// ====================
// Return TRUE if the Event Register of this PE is set, and FALSE if it is clear.

boolean IsEventRegisterSet()
    return EventRegister == '1';
```

G1.2.3.327 IsHighestEL

```
// IsHighestEL()
// =============
// Returns TRUE if given exception level is the highest exception level implemented

boolean IsHighestEL(bits(2) el)
    return HighestEL() == el;
```

G1.2.3.328 IsSecureBelowEL3

```
// IsSecureBelowEL3()
// ==================
// Return TRUE if an Exception level below EL3 is in Secure state
// or would be following an exception return to that level.
//
// That is, if at AArch64 EL3 or in AArch32 Monitor mode, whether an
// exception return would pass to Secure or Non-secure state.

boolean IsSecureBelowEL3()
    return TRUE;
```

G1.2.3.329 IsSecureEL2Enabled

```
// IsSecureEL2Enabled()
// ====================
// Returns TRUE if Secure EL2 is enabled, FALSE otherwise.

boolean IsSecureEL2Enabled()
    return TRUE;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.3.330 NonSecureOnlyImplementation

```
// NonSecureOnlyImplementation()
// =============================
// Returns TRUE if the security state is always Non-secure for this implementation.

boolean NonSecureOnlyImplementation()
    return boolean IMPLEMENTATION_DEFINED "Non-secure only implementation";
```

G1.2.3.331 PSTATE

```
// PSTATE
// ======
// Global per-processor state

ProcState PSTATE;
```

G1.2.3.332 PhysicalCountInt

```
// PhysicalCountInt()
// ==================
// Returns the integral part of physical count value of the System counter.

bits(64) PhysicalCountInt()
    return PhysicalCount<87:24>;
```

G1.2.3.333 PrivilegeLevel

```
// PrivilegeLevel
// ==============
// Privilege Level abstraction.

enumeration PrivilegeLevel {PL3, PL2, PL1, PL0};
```

G1.2.3.334 ProcState

```
// ProcState
// =========
// PE state bits.
// There is no significance to the field order.

type ProcState is (
    bits (1) N,        // Negative condition flag
    bits (1) Z,        // Zero condition flag
    bits (1) C,        // Carry condition flag
    bits (1) V,        // Overflow condition flag
    bits (1) D,        // Debug mask bit                      [AArch64 only]
    bits (1) A,        // SError interrupt mask bit
    bits (1) I,        // IRQ mask bit
    bits (1) F,        // FIQ mask bit
    bits (1) PAN,      // Privileged Access Never Bit         [v8.1]
    bits (1) UAO,      // User Access Override                [v8.2]
    bits (1) DIT,      // Data Independent Timing             [v8.4]
    bits (1) SS,       // Software step bit
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    bits (1) IL,       // Illegal Execution state bit
    bits (2) EL,       // Exception level
    bits (1) nRW,      // Execution state: 0=AArch64, 1=AArch32
    bits (1) SP,       // Stack pointer select: 0=SP0, 1=SPx [AArch64 only]
    bits (1) Q,        // Cumulative saturation flag         [AArch32 only]
    bits (4) GE,       // Greater than or Equal flags        [AArch32 only]
    bits (1) SSBS,     // Speculative Store Bypass Safe
    bits (8) IT,       // If-then bits, RES0 in CPSR         [AArch32 only]
    bits (1) J,        // J bit, RES0                        [AArch32 only, RES0 in SPSR and CPSR]
    bits (1) T,        // T32 bit, RES0 in CPSR              [AArch32 only]
    bits (1) E,        // Endianness bit                     [AArch32 only]
    bits (5) M         // Mode field                         [AArch32 only]
)
```

G1.2.3.335 SecureOnlyImplementation

```
// SecureOnlyImplementation()
// ==========================
// Returns TRUE if the security state is always Secure for this implementation.

boolean SecureOnlyImplementation()
    return boolean IMPLEMENTATION_DEFINED "Secure-only implementation";
```

G1.2.3.336 SecurityState

```
// SecurityState
// =============
// The Security state of an execution context

enumeration SecurityState {
    SS_NonSecure,
    SS_Secure
};
```

G1.2.3.337 SecurityStateAtEL

```
// SecurityStateAtEL()
// ===================
// Returns the effective security state at the exception level based off current settings.

SecurityState SecurityStateAtEL(bits(2) EL)
    return SS_Secure;
```

G1.2.3.338 SendEvent

```
// SendEvent()
// ===========
// Signal an event to all PEs in a multiprocessor system to set their Event Registers.
// When a PE executes the SEV instruction, it causes this function to be executed.

SendEvent();
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.3.339 SendEventLocal

```
// SendEventLocal()
// ================
// Set the local Event Register of this PE.
// When a PE executes the SEVL instruction, it causes this function to be executed.

SendEventLocal()
    EventRegister = '1';
    return;
```

G1.2.3.340 SetAccumulatedFPExceptions

```
// SetAccumulatedFPExceptions()
// ============================
// Stores FP Exceptions accumulated by the PE.

SetAccumulatedFPExceptions(bits(8) accumulated_exceptions);
```

G1.2.3.341 SetPSTATEFromPSR

```
// SetPSTATEFromPSR()
// ==================

SetPSTATEFromPSR(bits(N) spsr)
    constant boolean illegal_psr_state = IllegalExceptionReturn(spsr);
    SetPSTATEFromPSR(spsr, illegal_psr_state);

// SetPSTATEFromPSR()
// ==================
// Set PSTATE based on a PSR value

SetPSTATEFromPSR(bits(N) spsr_in, boolean illegal_psr_state)
    bits(N) spsr = spsr_in;
    constant boolean from_aarch64 = !UsingAArch32();
    PSTATE.SS = DebugExceptionReturnSS(spsr);

    ShouldAdvanceSS = FALSE;
    if illegal_psr_state then
        PSTATE.IL = '1';
        if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = bit UNKNOWN;
        if IsFeatureImplemented(FEAT_UAO) then PSTATE.UAO = bit UNKNOWN;
        if IsFeatureImplemented(FEAT_DIT) then PSTATE.DIT = bit UNKNOWN;
    else
        // State that is reinstated only on a legal exception return
        PSTATE.IL = spsr<20>;
        PSTATE.nRW = '0';
        (-, PSTATE.EL) = ELFromSPSR(spsr);
        PSTATE.SP  = spsr<0>;
        if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = spsr<12>;
        if IsFeatureImplemented(FEAT_UAO) then PSTATE.UAO = spsr<23>;
        if IsFeatureImplemented(FEAT_DIT) then PSTATE.DIT = spsr<24>;

    // State that is reinstated regardless of illegal exception return
    PSTATE.<N,Z,C,V> = spsr<31:28>;
    if IsFeatureImplemented(FEAT_PAN) then PSTATE.PAN = spsr<22>;
    PSTATE.<D,A,I,F> = spsr<9:6>;
    return;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.3.342 ShouldAdvanceHS

```
// ShouldAdvanceHS
// ===============
// Cleared if we should not advance the EDESR.SS after the current instruction.

boolean ShouldAdvanceHS;
```

G1.2.3.343 ShouldAdvanceSS

```
// ShouldAdvanceSS
// ===============
// Cleared if PSTATE.SS is written by the current instruction.

boolean ShouldAdvanceSS;
```

G1.2.3.344 SmallestTranslationGranule

```
// SmallestTranslationGranule()
// ============================
// Smallest implemented translation granule.

integer SmallestTranslationGranule()
    if IsFeatureImplemented(FEAT_TGran4K) then return 12;
    if IsFeatureImplemented(FEAT_TGran16K) then return 14;
    if IsFeatureImplemented(FEAT_TGran64K)  then return 16;
    Unreachable();
```

G1.2.3.345 SpeculationBarrier

```
// SpeculationBarrier()
// ====================

SpeculationBarrier();
```

G1.2.3.346 SynchronizeContext

```
// SynchronizeContext()
// ====================
// Context Synchronization event, includes Instruction Fetch Barrier effect

SynchronizeContext();
```

G1.2.3.347 SynchronizeErrors

```
// SynchronizeErrors()
// ===================
// Implements the error synchronization event.

SynchronizeErrors();
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.3.348 TakeUnmaskedPhysicalSErrorInterrupts

```
// TakeUnmaskedPhysicalSErrorInterrupts()
// ======================================
// Take any pending unmasked physical SError interrupt.

TakeUnmaskedPhysicalSErrorInterrupts(boolean iesb_req);
```

G1.2.3.349 TakeUnmaskedSErrorInterrupts

```
// TakeUnmaskedSErrorInterrupts()
// ==============================
// Take any pending unmasked physical SError interrupt or unmasked virtual SError
// interrupt.

TakeUnmaskedSErrorInterrupts();
```

G1.2.3.350 ThisInstr

```
// ThisInstr()
// ===========

bits(32) ThisInstr();
```

G1.2.3.351 ThisInstrLength

```
// ThisInstrLength()
// =================

integer ThisInstrLength();
```

G1.2.3.352 UsingAArch32

```
// UsingAArch32()
// ==============
// Return TRUE if the current Exception level is using AArch32, FALSE if using AArch64.

boolean UsingAArch32()
    constant boolean aarch32 = (PSTATE.nRW == '1');
    if !HaveAArch64() then assert aarch32;
    return aarch32;
```

G1.2.3.353 ValidSecurityStateAtEL

```
// ValidSecurityStateAtEL()
// ========================
// Returns TRUE if the current settings and architecture choices for this
// implementation permit a valid Security state at the indicated EL.

boolean ValidSecurityStateAtEL(bits(2) el)
    if !HaveEL(el) then
        return FALSE;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if el == EL2 then
        return EL2Enabled();

    return TRUE;
```

G1.2.3.354 VirtualFIQPending

```
// VirtualFIQPending()
// ===================
// Returns TRUE if there is any pending virtual FIQ.

boolean VirtualFIQPending();
```

G1.2.3.355 VirtualIRQPending

```
// VirtualIRQPending()
// ===================
// Returns TRUE if there is any pending virtual IRQ.

boolean VirtualIRQPending();
```

G1.2.3.356 WFxType

```
// WFxType
// =======
// WFx instruction types.

enumeration WFxType {WFxType_WFE, WFxType_WFI};
```

G1.2.3.357 WaitForEvent

```
// WaitForEvent()
// ==============
// PE optionally suspends execution until one of the following occurs:
// - A WFE wakeup event.
// - A reset.
// - The implementation chooses to resume execution.
// It is IMPLEMENTATION DEFINED whether restarting execution after the period of
// suspension causes the Event Register to be cleared.

WaitForEvent()
    if Halted() then return;
    if !IsEventRegisterSet() then
        EnterLowPowerState();
    return;
```

G1.2.3.358 WaitForInterrupt

```
// WaitForInterrupt()
// ==================
// PE optionally suspends execution until one of the following occurs:
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// - A WFI wakeup event.
// - A reset.
// - The implementation chooses to resume execution.

WaitForInterrupt()
    if Halted() then return;
    EnterLowPowerState();
    return;
```

G1.2.3.359 WatchpointRelatedSyndrome

```
// WatchpointRelatedSyndrome()
// ===========================
// Update common Watchpoint related fields.

bits(24) WatchpointRelatedSyndrome(FaultRecord fault)
    bits(24) syndrome = Zeros(24);

    if IsFeatureImplemented(FEAT_Debugv8p2) then
        if syndrome<15> == '1' || syndrome<10> == '1' then           // Either of FnP or FnV is 1
            syndrome<17> = '1';                                      // WPTV
        elsif !AddressInNaturallyAlignedBlock(fault.vaddress, fault.watchptinfo.vaddress) then
            syndrome<17> = '1';                                      // WPTV
        elsif boolean IMPLEMENTATION_DEFINED "WPTV field is valid" then
            syndrome<17> = '1';
        if syndrome<17> == '1' then
            syndrome<23:18> = fault.watchptinfo.watchpt_num<5:0>;    // WPT
        else
            syndrome<23:18> = bits(6) UNKNOWN;

    return syndrome;
```

G1.2.3.360 ASID_NONE

```
constant bits(16) ASID_NONE = Zeros(16);
```

G1.2.3.361 Broadcast

```
// Broadcast
// =========

enumeration Broadcast {
    Broadcast_ISH,
    Broadcast_ForcedISH,
    Broadcast_OSH,
    Broadcast_NSH
};
```

G1.2.3.362 BroadcastTLBI

```
// BroadcastTLBI()
// ===============
// IMPLEMENTATION DEFINED function to broadcast TLBI operation within the indicated broadcast
// domain.
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
BroadcastTLBI(Broadcast broadcast, TLBIRecord r)
    IMPLEMENTATION_DEFINED;
```

G1.2.3.363 TLBI

```
// TLBI()
// ======
// Invalidates TLB entries for which TLBIMatch() returns TRUE.

TLBI(TLBIRecord r)
    IMPLEMENTATION_DEFINED;
```

G1.2.3.364 TLBILevel

```
// TLBILevel
// =========

enumeration TLBILevel {
    TLBILevel_Any,        // this applies to TLB entries at all levels
    TLBILevel_Last        // this applies to TLB entries at last level only
};
```

G1.2.3.365 TLBIMemAttr

```
// TLBIMemAttr
// ===========
// Defines the attributes of the memory operations that must be completed in
// order to deem the TLBI operation as completed.

enumeration TLBIMemAttr {
    TLBI_AllAttr,         // All TLB entries within the scope of the invalidation
    TLBI_ExcludeXS        // Only TLB entries with XS=0 within the scope of the invalidation
};
```

G1.2.3.366 TLBIOp

```
// TLBIOp
// ======

enumeration TLBIOp {
    TLBIOp_ALL,
    TLBIOp_ASID,
    TLBIOp_IPAS2,
    TLBIOp_VAA,
    TLBIOp_VA,
    TLBIOp_VMALL,
    TLBIOp_VMALLS12,
    TLBIOp_RIPAS2,
    TLBIOp_RVAA,
    TLBIOp_RVA,
};
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.3.367 TLBIRecord

```
// TLBIRecord
// ==========
// Details related to a TLBI operation.

type TLBIRecord is (
    TLBIOp          op,
    boolean         from_aarch64, // originated as an AArch64 operation
    SecurityState   security,
    Regime          regime,
    boolean         use_vmid,
    bits(16)        vmid,
    bits(16)        asid,
    TLBILevel       level,
    TLBIMemAttr     attr,
    PASpace         ipaspace,     // For operations that take IPA as input address
    bits(64)        address,      // input address, for range operations, start address
    bits(64)        end_address,  // for range operations, end address
    bits(4)         ttl,          // translation table walk level holding the leaf entry
                                  // for the address being invalidated
    bits(2)         tg            // for range operations, translation granule
)
```

G1.2.3.368 VMID

```
// VMID - getter
// =============
// Effective VMID.

bits(16) VMID[]
    if !IsFeatureImplemented(FEAT_VMID16) || VTCR_EL2.VS == '0' then
        return ZeroExtend(VSCTLR_EL2.VMID<7:0>, 16);
    else
        return VSCTLR_EL2.VMID;
```

G1.2.3.369 VMID_NONE

```
constant bits(16) VMID_NONE = Zeros(16);
```

G1.2.3.370 ConstrainUnpredictable

```
// ConstrainUnpredictable()
// ========================
// Return the appropriate Constraint result to control the caller's behavior.
// The return value is IMPLEMENTATION DEFINED within a permitted list for each
// UNPREDICTABLE case.
// (The permitted list is determined by an assert or case statement at the call site.)

Constraint ConstrainUnpredictable(Unpredictable which);
```

G1.2.3.371 ConstrainUnpredictableBits

```
// ConstrainUnpredictableBits()
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// ============================

// This is a variant of ConstrainUnpredictable for when the result can be Constraint_UNKNOWN.
// If the result is Constraint_UNKNOWN then the function also returns UNKNOWN value, but that
// value is always an allocated value; that is, one for which the behavior is not itself
// CONSTRAINED.

(Constraint,bits(width)) ConstrainUnpredictableBits(Unpredictable which, integer width);
```

G1.2.3.372 ConstrainUnpredictableBool

```
// ConstrainUnpredictableBool()
// ============================
// This is a variant of the ConstrainUnpredictable function where the result is either
// Constraint_TRUE or Constraint_FALSE.

boolean ConstrainUnpredictableBool(Unpredictable which);
```

G1.2.3.373 ConstrainUnpredictableInteger

```
// ConstrainUnpredictableInteger()
// ===============================
// This is a variant of ConstrainUnpredictable for when the result can be Constraint_UNKNOWN.
// If the result is Constraint_UNKNOWN then the function also returns an UNKNOWN
// value in the range low to high, inclusive.

(Constraint,integer) ConstrainUnpredictableInteger(integer low, integer high,
                                                   Unpredictable which);
```

G1.2.3.374 ConstrainUnpredictableProcedure

```
// ConstrainUnpredictableProcedure()
// =================================
// This is a variant of ConstrainUnpredictable that implements a Constrained
// Unpredictable behavior for a given Unpredictable situation.
// The behavior is within permitted behaviors for a given Unpredictable situation,
// these are documented in the textual part of the architecture specification.
//
// This function is expected to be refined in an IMPLEMENTATION DEFINED manner.
// The details of possible outcomes may not be present in the code and must be interpreted
// for each use with respect to the CONSTRAINED UNPREDICTABLE specifications
// for the specific area.

ConstrainUnpredictableProcedure(Unpredictable which);
```

G1.2.3.375 Constraint

```
// Constraint
// ==========
// List of Constrained Unpredictable behaviors.

enumeration Constraint    {// General
                           Constraint_NONE,             // Instruction executes with
                                                        // no change or side-effect
                                                        // to its described behavior
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                              Constraint_UNKNOWN,           // Destination register
                                                            // has UNKNOWN value
                              Constraint_UNDEF,             // Instruction is UNDEFINED
                              Constraint_UNDEFEL0,          // Instruction is UNDEFINED at EL0 only
                              Constraint_NOP,               // Instruction executes as NOP
                              Constraint_TRUE,
                              Constraint_FALSE,
                              Constraint_DISABLED,
                              Constraint_UNCOND,            // Instruction executes unconditionally
                              Constraint_COND,              // Instruction executes conditionally
                              Constraint_ADDITIONAL_DECODE, // Instruction executes
                                                            // with additional decode
                              // Load-store
                              Constraint_WBSUPPRESS,
                              Constraint_FAULT,
                              Constraint_MPU_FAULT,         // Raise MPU fault
                              Constraint_MPU_ATTR_UNKNOWN,  // MPU Attribute is UNKNOWN
                              Constraint_EL1TIMESTAMP,      // Constrain to Virtual Timestamp
                              Constraint_EL2TIMESTAMP,      // Constrain to Virtual Timestamp
                              Constraint_OSH,               // Constrain to Outer Shareable
                              Constraint_ISH,               // Constrain to Inner Shareable
                              Constraint_NSH,               // Constrain to Nonshareable

                              Constraint_NC,                // Constrain to Noncacheable
                              Constraint_WT,                // Constrain to Writethrough
                              Constraint_WB,                // Constrain to Writeback

                              // IPA too large
                              Constraint_FORCE, Constraint_FORCENOSLCHECK,
                              // PMSCR_PCT reserved values select Virtual timestamp
                              Constraint_PMSCR_PCT_VIRT,
                              Constraint_ANYREG,            // Any allocated register is chosen
};
```

G1.2.3.376 Unpredictable

```
// Unpredictable
// =============
// List of Constrained Unpredictable situations.

enumeration Unpredictable {
                              // VMSR on MVFR
                              Unpredictable_VMSR,
                              // Writeback/transfer register overlap (load)
                              Unpredictable_WBOVERLAPLD,
                              // Writeback/transfer register overlap (store)
                              Unpredictable_WBOVERLAPST,
                              // Load Pair transfer register overlap
                              Unpredictable_LDPOVERLAP,
                              // Store-exclusive base/status register overlap
                              Unpredictable_BASEOVERLAP,
                              // Store-exclusive data/status register overlap
                              Unpredictable_DATAOVERLAP,
                              // Load-store alignment checks
                              Unpredictable_DEVPAGE2,
                              // Instruction fetch from Device memory
                              Unpredictable_INSTRDEVICE,
                              // Reserved CPACR value
                              Unpredictable_RESCPACR,
                              // Reserved MAIR value
                              Unpredictable_RESMAIR,
                              // Reserved Stage 2 MemAttr value
                              Unpredictable_S2RESMEMATTR,
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    // Reserved TEX:C:B value
    Unpredictable_RESTEXCB,
    // Reserved PRRR value
    Unpredictable_RESPRRR,
    // Reserved DACR field
    Unpredictable_RESDACR,
    // Reserved VTCR.S value
    Unpredictable_RESVTCRS,
    // Reserved TCR.TnSZ value
    Unpredictable_RESTnSZ,
    // Out-of-range TCR.TnSZ value
    Unpredictable_OORTnSZ,

    // IPA size exceeds PA size
    Unpredictable_LARGEIPA,
    // Syndrome for a known-passing conditional A32 instruction
    Unpredictable_ESRCONDPASS,
    // Illegal State exception: zero PSTATE.IT
    Unpredictable_ILZEROIT,
    // Illegal State exception: zero PSTATE.T
    Unpredictable_ILZEROT,
    // Debug: prioritization of Vector Catch
    Unpredictable_BPVECTORCATCHPRI,
    // Debug Vector Catch: match on 2nd halfword
    Unpredictable_VCMATCHHALF,
    // Debug Vector Catch: match on Data Abort
    // or Prefetch abort
    Unpredictable_VCMATCHDAPA,
    // Debug watchpoints: nonzero MASK and non-ones BAS
    Unpredictable_WPMASKANDBAS,
    // Debug watchpoints: non-contiguous BAS
    Unpredictable_WPBASCONTIGUOUS,
    // Debug watchpoints: reserved MASK
    Unpredictable_RESWPMASK,
    // Debug watchpoints: nonzero MASKed bits of address
    Unpredictable_WPMASKEDBITS,
    // Debug breakpoints and watchpoints: reserved control bits
    Unpredictable_RESBPWPCTRL,
    // Debug breakpoints: not implemented
    Unpredictable_BPNOTIMPL,
    // Debug breakpoints: reserved type
    Unpredictable_RESBPTYPE,
    // Debug breakpoints: not-context-aware breakpoint
    Unpredictable_BPNOTCTXCMP,
    // Debug breakpoints: match on 2nd halfword of instruction
    Unpredictable_BPMATCHHALF,
    // Debug breakpoints: mismatch on 2nd halfword of instruction
    Unpredictable_BPMISMATCHHALF,
    // Debug breakpoints: a breakpoint is linked to that is not
    // programmed with linking enabled
    Unpredictable_BPLINKINGDISABLED,
    // Debug: restart to a misaligned AArch32 PC value
    Unpredictable_RESTARTALIGNPC,
    // Debug: restart to a not-zero-extended AArch32 PC value
    Unpredictable_RESTARTZEROUPPERPC,
    // Zero top 32 bits of X registers in AArch32 state
    Unpredictable_ZEROUPPER,
    // Zero top 32 bits of PC on illegal return to
    // AArch32 state
    Unpredictable_ERETZEROUPPERPC,
    // Force address to be aligned when interworking
    // branch to A32 state
    Unpredictable_A32FORCEALIGNPC,
    // SMC disabled
    Unpredictable_SMD,
    // Stage-2 EL1&0 mismatch MPU configuration.
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                                Unpredictable_S2EL1MPUCONFIG,
                                // Reserved shareability encoding
                                Unpredictable_Shareability,
                                // Access Flag Update by HW
                                Unpredictable_AFUPDATE,
                                // Dirty Bit State Update by HW
                                Unpredictable_DBUPDATE,
                                // Unsupported memory attributes for HW update
                                Unpredictable_Unsupported_Atomic_HW_Update,
                                // Consider SCTLR_ELx[].IESB in Debug state
                                Unpredictable_IESBinDebug,
                                // Timestamp constrained to virtual or physical
                                Unpredictable_EL2TIMESTAMP,
                                Unpredictable_EL1TIMESTAMP,
                                // Unaligned exclusives, atomics, acquire/release
                                // to a region that is not to Normal inner write-back
                                // outer write-back shareable generate an Alignment fault.
                                Unpredictable_LSE2_ALIGNMENT_FAULT,
                                // Clearing DCC/ITR sticky flags when instruction is in flight
                                Unpredictable_CLEARERRITEZERO,
                                // ALUEXCEPTIONRETURN when in user/system mode in
                                // A32 instructions
                                Unpredictable_ALUEXCEPTIONRETURN,
                                // Trap to register in debug state are ignored
                                Unpredictable_IGNORETRAPINDEBUG,
                                // Compare DBGBVR.RESS for BP/WP
                                Unpredictable_DBGxVR_RESS,
                                // Inaccessible event counter
                                Unpredictable_PMUEVENTCOUNTER,
                                // Reserved PMSCR.PCT behavior
                                Unpredictable_PMSCR_PCT,
                                // MDCR_EL2.HPMN or HDCR.HPMN is larger than PMCR.N or
                                // FEAT_HPMN0 is not implemented and HPMN is 0.
                                Unpredictable_RES_HPMN,
                                // Chained PMU counters idx and idx+1 are not in same range
                                Unpredictable_COUNT_CHAIN,
                                // Chooses which value to return in a non failed Atomic Compare and Swap
                                Unpredictable_CASRETURNOLDVALUE,
                                // Enables write of the newvalue in a failed Atomic Compare and Swap
                                Unpredictable_WRITEFAILEDCAS,
                                // Any access to MPU region register above the number of
                                // implemented regions specified by
                                // MPUIR_ELx.REGION/H(MPUIR).REGION is constrained unpredictable
                                Unpredictable_UnimplementedMPURegion_RW_UNKNOWN,
                                // If the atomic instructions are not atomic in regard to other
                                // agents that access memory, then the instruction can have one or
                                // more of the following effects
                                Unpredictable_Atomic_SYNC_ABORT,
                                Unpredictable_Atomic_SERROR,
                                Unpredictable_Atomic_NOP,
                                Unpredictable_Atomic_MMU_IMPDEF_FAULT,
                                // Accessing DBGDSCRint via MRC in debug state
                                Unpredictable_MRC_APSR_TARGET,
};
```

G1.2.3.377 AdvSIMDExpandImm

```
// AdvSIMDExpandImm()
// ==================

bits(64) AdvSIMDExpandImm(bit op, bits(4) cmode, bits(8) imm8)
    bits(64) imm64;
    case cmode<3:1> of
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when '000'
            imm64 = Replicate(Zeros(24):imm8, 2);
        when '001'
            imm64 = Replicate(Zeros(16):imm8:Zeros(8), 2);
        when '010'
            imm64 = Replicate(Zeros(8):imm8:Zeros(16), 2);
        when '011'
            imm64 = Replicate(imm8:Zeros(24), 2);
        when '100'
            imm64 = Replicate(Zeros(8):imm8, 4);
        when '101'
            imm64 = Replicate(imm8:Zeros(8), 4);
        when '110'
            if cmode<0> == '0' then
                imm64 = Replicate(Zeros(16):imm8:Ones(8), 2);
            else
                imm64 = Replicate(Zeros(8):imm8:Ones(16), 2);
        when '111'
            if cmode<0> == '0' && op == '0' then
                imm64 = Replicate(imm8, 8);
            if cmode<0> == '0' && op == '1' then
                imm8a = Replicate(imm8<7>, 8); imm8b = Replicate(imm8<6>, 8);
                imm8c = Replicate(imm8<5>, 8); imm8d = Replicate(imm8<4>, 8);
                imm8e = Replicate(imm8<3>, 8); imm8f = Replicate(imm8<2>, 8);
                imm8g = Replicate(imm8<1>, 8); imm8h = Replicate(imm8<0>, 8);
                imm64 = imm8a:imm8b:imm8c:imm8d:imm8e:imm8f:imm8g:imm8h;
            if cmode<0> == '1' && op == '0' then
                imm32 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>, 5):imm8<5:0>:Zeros(19);
                imm64 = Replicate(imm32, 2);
            if cmode<0> == '1' && op == '1' then
                if UsingAArch32() then ReservedEncoding();
                imm64 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>, 8):imm8<5:0>:Zeros(48);

    return imm64;
```

G1.2.3.378 PolynomialMult

```
// PolynomialMult()
// ================

bits(M+N) PolynomialMult(bits(M) op1, bits(N) op2)
    result = Zeros(M+N);
    extended_op2 = ZeroExtend(op2, M+N);
    for i=0 to M-1
        if op1<i> == '1' then
            result = result EOR LSL(extended_op2, i);
    return result;
```

G1.2.3.379 SatQ

```
// SatQ()
// ======

(bits(N), boolean) SatQ(integer i, integer N, boolean unsigned)
    (result, sat) = if unsigned then UnsignedSatQ(i, N) else SignedSatQ(i, N);
    return (result, sat);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.3.380 ShiftSat

```
// ShiftSat()
// ==========

integer ShiftSat(integer shift, integer esize)
    if shift > esize+1 then return esize+1;
    elsif shift < -(esize+1) then return -(esize+1);
    return shift;
```

G1.2.3.381 SignedSat

```
// SignedSat()
// ===========

bits(N) SignedSat(integer i, integer N)
    (result, -) = SignedSatQ(i, N);
    return result;
```

G1.2.3.382 SignedSatQ

```
// SignedSatQ()
// ============

(bits(N), boolean) SignedSatQ(integer i, integer N)
    integer result;
    boolean saturated;
    if i > 2^(N-1) - 1 then
        result = 2^(N-1) - 1;  saturated = TRUE;
    elsif i < -(2^(N-1)) then
        result = -(2^(N-1));  saturated = TRUE;
    else
        result = i;  saturated = FALSE;
    return (result<N-1:0>, saturated);
```

G1.2.3.383 UnsignedRSqrtEstimate

```
// UnsignedRSqrtEstimate()
// =======================

bits(N) UnsignedRSqrtEstimate(bits(N) operand)
    assert N == 32;
    bits(N) result;
    if operand<N-1:N-2> == '00' then  // Operands <= 0x3FFFFFFF produce 0xFFFFFFFF
        result = Ones(N);
    else
        // input is in the range 0x40000000 .. 0xffffffff representing [0.25 .. 1.0)
        // estimate is in the range 256 .. 511 representing [1.0 .. 2.0)
        estimate = RecipSqrtEstimate(UInt(operand<31:23>));
        // result is in the range 0x80000000 .. 0xff800000 representing [1.0 .. 2.0)
        result = estimate<8:0> : Zeros(N-9);

    return result;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.3.384 UnsignedRecipEstimate

```
// UnsignedRecipEstimate()
// =======================

bits(N) UnsignedRecipEstimate(bits(N) operand)
    assert N == 32;
    bits(N) result;
    if operand<N-1> == '0' then  // Operands <= 0x7FFFFFFF produce 0xFFFFFFFF
        result = Ones(N);
    else
        // input is in the range 0x80000000 .. 0xffffffff representing [0.5 .. 1.0)

        // estimate is in the range 256 to 511 representing [1.0 .. 2.0)
        estimate = RecipEstimate(UInt(operand<31:23>));

        // result is in the range 0x80000000 .. 0xff800000 representing [1.0 .. 2.0)
        result = estimate<8:0> : Zeros(N-9);

    return result;
```

G1.2.3.385 UnsignedSat

```
// UnsignedSat()
// =============

bits(N) UnsignedSat(integer i, integer N)
    (result, -) = UnsignedSatQ(i, N);
    return result;
```

G1.2.3.386 UnsignedSatQ

```
// UnsignedSatQ()
// ==============

(bits(N), boolean) UnsignedSatQ(integer i, integer N)
    integer result;
    boolean saturated;
    if i > 2^N - 1 then
        result = 2^N - 1;  saturated = TRUE;
    elsif i < 0 then
        result = 0;  saturated = TRUE;
    else
        result = i;  saturated = FALSE;
    return (result<N-1:0>, saturated);
```

G1.2.4 shared/trace

This section includes the following pseudocode functions:

G1.2.4.1 GetTimestamp

```
// GetTimestamp()
// ==============
// Returns the Timestamp depending on the type
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
bits(64) GetTimestamp(TimeStamp timeStampType)
    case timeStampType of
        when TimeStamp_Physical
            return PhysicalCountInt();
        when TimeStamp_Virtual
            return PhysicalCountInt() - CNTVOFF_EL2;
        when TimeStamp_None
            return Zeros(64);
        when TimeStamp_CoreSight
            return bits(64) IMPLEMENTATION_DEFINED "CoreSight timestamp";
        otherwise
            Unreachable();
```

G1.2.4.2 EffectiveE0HTRE

```
// EffectiveE0HTRE()
// =================
// Returns effective E0HTRE value

bit EffectiveE0HTRE()
    return TRFCR_EL2.E0HTRE;
```

G1.2.4.3 EffectiveE0TRE

```
// EffectiveE0TRE()
// ================
// Returns effective E0TRE value

bit EffectiveE0TRE()
    return TRFCR_EL1.E0TRE;
```

G1.2.4.4 EffectiveE1TRE

```
// EffectiveE1TRE()
// ================
// Returns effective E1TRE value

bit EffectiveE1TRE()
    return TRFCR_EL1.E1TRE;
```

G1.2.4.5 EffectiveE2TRE

```
// EffectiveE2TRE()
// ================
// Returns effective E2TRE value

bit EffectiveE2TRE()
    return TRFCR_EL2.E2TRE;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.4.6 SelfHostedTraceEnabled

```
// SelfHostedTraceEnabled()
// ========================
// Returns TRUE if Self-hosted Trace is enabled.

boolean SelfHostedTraceEnabled()
    bit secure_trace_enable = '0';
    if !(HaveTraceExt() && IsFeatureImplemented(FEAT_TRF)) then return FALSE;
    if EDSCR.TFO == '0' then return TRUE;
    return !ExternalSecureNoninvasiveDebugEnabled();
```

G1.2.4.7 TraceAllowed

```
// TraceAllowed()
// ==============
// Returns TRUE if Self-hosted Trace is allowed in the given Exception level.

boolean TraceAllowed(bits(2) el)
    if !HaveTraceExt() then
        return FALSE;
    // If in Debug state then tracing is not allowed
    if Halted() && !Restarting() then
        return FALSE;
    if SelfHostedTraceEnabled() then
        boolean trace_allowed;
        ss = SecurityStateAtEL(el);
        bit TRE_bit;
        case el of
            when EL3  TRE_bit = '0';
            when EL2  TRE_bit = EffectiveE2TRE();
            when EL1  TRE_bit = EffectiveE1TRE();
            when EL0
                if EffectiveTGE() == '1' then
                    TRE_bit = EffectiveE0HTRE();
                else
                    TRE_bit = EffectiveE0TRE();

        return trace_allowed && TRE_bit == '1';
    else
        return ExternalNoninvasiveDebugAllowed(el);
```

G1.2.4.8 TraceContextIDR2

```
// TraceContextIDR2()
// ==================

boolean TraceContextIDR2()
    if !TraceAllowed(PSTATE.EL)|| !HaveEL(EL2) then return FALSE;
    return (!SelfHostedTraceEnabled() || TRFCR_EL2.CX == '1');
```

G1.2.4.9 TraceSynchronizationBarrier

```
// TraceSynchronizationBarrier()
// =============================
// Barrier instruction that preserves the relative order of accesses to System
// registers due to trace operations and other accesses to the same registers.
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TraceSynchronizationBarrier();
```

G1.2.4.10 TraceTimeStamp

```
// TraceTimeStamp()
// ================

TimeStamp TraceTimeStamp()
    if SelfHostedTraceEnabled() then
        if HaveEL(EL2) then
            TS_el2 = TRFCR_EL2.TS;
            if TS_el2 == '10' then
                // Reserved value
                (-, TS_el2) = ConstrainUnpredictableBits(Unpredictable_EL2TIMESTAMP, 2);

            case TS_el2 of
                when '00'
                    // Falls out to check TRFCR_EL1.TS
                when '01'
                    return TimeStamp_Virtual;
                when '11'
                    return TimeStamp_Physical;

        TS_el1 = TRFCR_EL1.TS;
        if TS_el1 == 'x0' then
            // Reserved value
            (-, TS_el1) = ConstrainUnpredictableBits(Unpredictable_EL1TIMESTAMP, 2);

        case TS_el1 of
            when '01'
                return TimeStamp_Virtual;
            when '11'
                return TimeStamp_Physical;
            otherwise
                Unreachable();         // ConstrainUnpredictableBits removes this case
    else
        return TimeStamp_CoreSight;
```

G1.2.5 shared/translation

This section includes the following pseudocode functions:

G1.2.5.1 at

```
enumeration TranslationStage {
    TranslationStage_1,
    TranslationStage_12
};

enumeration ATAccess {
    ATAccess_Read,
    ATAccess_Write,
    ATAccess_ReadPAN,
    ATAccess_WritePAN
};
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.5.2 EncodePARAttrs

```
// EncodePARAttrs()
// ================
// Convert orthogonal attributes and hints to 64-bit PAR ATTR field.

bits(8) EncodePARAttrs(MemoryAttributes memattrs)
    bits(8) result;

    if memattrs.memtype == MemType_Device then
        result<7:4> = '0000';
        case memattrs.device of
            when DeviceType_nGnRnE result<3:0> = '0000';
            when DeviceType_nGnRE  result<3:0> = '0100';
            when DeviceType_nGRE   result<3:0> = '1000';
            when DeviceType_GRE    result<3:0> = '1100';
            otherwise              Unreachable();
    else
        if memattrs.outer.attrs == MemAttr_WT then
            result<7:6> = if memattrs.outer.transient then '00' else '10';
            result<5:4> = memattrs.outer.hints;
        elsif memattrs.outer.attrs == MemAttr_WB then
            result<7:6> = if memattrs.outer.transient then '01' else '11';
            result<5:4> = memattrs.outer.hints;
        else // MemAttr_NC
            result<7:4> = '0100';

        if memattrs.inner.attrs == MemAttr_WT then
            result<3:2> = if memattrs.inner.transient then '00' else '10';
            result<1:0> = memattrs.inner.hints;
        elsif memattrs.inner.attrs == MemAttr_WB then
            result<3:2> = if memattrs.inner.transient then '01' else '11';
            result<1:0> = memattrs.inner.hints;
        else // MemAttr_NC
            result<3:0> = '0100';

    return result;
```

G1.2.5.3 PAREncodeShareability

```
// PAREncodeShareability()
// =======================
// Derive 64-bit PAR SH field.

bits(2) PAREncodeShareability(MemoryAttributes memattrs)
    if (memattrs.memtype == MemType_Device ||
            (memattrs.inner.attrs == MemAttr_NC &&
             memattrs.outer.attrs == MemAttr_NC)) then
        // Force Outer-Shareable on Device and Normal Non-Cacheable memory
        return '10';

    case memattrs.shareability of
        when Shareability_NSH return '00';
        when Shareability_ISH return '11';
        when Shareability_OSH return '10';
```

G1.2.5.4 ReportedPARAttrs

```
// ReportedPARAttrs()
// ==================
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// The value returned in this field can be the resulting attribute, as determined by any permitted
// implementation choices and any applicable configuration bits, instead of the value that appears
// in the translation table descriptor.

bits(8) ReportedPARAttrs(bits(8) parattrs);
```

G1.2.5.5 ReportedPARShareability

```
// ReportedPARShareability()
// =========================
// The value returned in SH field can be the resulting attribute, as determined by any
// permitted implementation choices and any applicable configuration bits, instead of
// the value that appears in the translation table descriptor.

bits(2) ReportedPARShareability(bits(2) sh);
```

G1.2.5.6 DecodeDevice

```
// DecodeDevice()
// ==============
// Decode output Device type

DeviceType DecodeDevice(bits(2) device)
    case device of
        when '00'   return DeviceType_nGnRnE;
        when '01'   return DeviceType_nGnRE;
        when '10'   return DeviceType_nGRE;
        when '11'   return DeviceType_GRE;
```

G1.2.5.7 DecodeLDFAttr

```
// DecodeLDFAttr()
// ===============
// Decode memory attributes using LDF (Long Descriptor Format) mapping

MemAttrHints DecodeLDFAttr(bits(4) attr)
    MemAttrHints ldfattr;

    if    attr == 'x0xx' then ldfattr.attrs = MemAttr_WT; // Write-through
    elsif attr == '0100' then ldfattr.attrs = MemAttr_NC; // Non-cacheable
    elsif attr == 'x1xx' then ldfattr.attrs = MemAttr_WB; // Write-back
    else                      Unreachable();

    // Allocation hints are applicable only to cacheable memory.
    if ldfattr.attrs != MemAttr_NC then
        case attr<1:0> of
            when '00' ldfattr.hints = MemHint_No;  // No allocation hints
            when '01' ldfattr.hints = MemHint_WA;  // Write-allocate
            when '10' ldfattr.hints = MemHint_RA;  // Read-allocate
            when '11' ldfattr.hints = MemHint_RWA; // Read/Write allocate

    // The Transient hint applies only to cacheable memory with some allocation hints.
    if ldfattr.attrs != MemAttr_NC && ldfattr.hints != MemHint_No then
        ldfattr.transient = attr<3> == '0';

    return ldfattr;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.5.8 DecodeSDFAttr

```
// DecodeSDFAttr()
// ===============
// Decode memory attributes using SDF (Short Descriptor Format) mapping

MemAttrHints DecodeSDFAttr(bits(2) rgn)
    MemAttrHints sdfattr;

    case rgn of
        when '00'                   // Non-cacheable (no allocate)
            sdfattr.attrs = MemAttr_NC;
        when '01'                   // Write-back, Read and Write allocate
            sdfattr.attrs = MemAttr_WB;
            sdfattr.hints = MemHint_RWA;
        when '10'                   // Write-through, Read allocate
            sdfattr.attrs = MemAttr_WT;
            sdfattr.hints = MemHint_RA;
        when '11'                   // Write-back, Read allocate
            sdfattr.attrs = MemAttr_WB;
            sdfattr.hints = MemHint_RA;

    sdfattr.transient = FALSE;

    return sdfattr;
```

G1.2.5.9 DecodeShareability

```
// DecodeShareability()
// ====================
// Decode shareability of target memory region

Shareability DecodeShareability(bits(2) sh)
    case sh of
        when '10' return Shareability_OSH;
        when '11' return Shareability_ISH;
        when '00' return Shareability_NSH;
        otherwise
            case ConstrainUnpredictable(Unpredictable_Shareability) of
                when Constraint_OSH return Shareability_OSH;
                when Constraint_ISH return Shareability_ISH;
                when Constraint_NSH return Shareability_NSH;
```

G1.2.5.10 EffectiveShareability

```
// EffectiveShareability()
// =======================
// Force Outer Shareability on Device and Normal iNCoNC memory

Shareability EffectiveShareability(MemoryAttributes memattrs)
    if (memattrs.memtype == MemType_Device ||
            (memattrs.inner.attrs == MemAttr_NC &&
             memattrs.outer.attrs == MemAttr_NC)) then
        return Shareability_OSH;
    else
        return memattrs.shareability;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.5.11 IsWBShareable

```
// IsWBShareable()
// ===============
// Determines whether the given memory attributes are iWBoWB Shareable

boolean IsWBShareable(MemoryAttributes memattrs)

    return (memattrs.memtype == MemType_Normal &&
        memattrs.inner.attrs == MemAttr_WB &&
        memattrs.outer.attrs == MemAttr_WB &&
        memattrs.shareability IN {Shareability_ISH, Shareability_OSH});
```

G1.2.5.12 NormalNCMemAttr

```
// NormalNCMemAttr()
// =================
// Normal Non-cacheable memory attributes

MemoryAttributes NormalNCMemAttr()
    MemAttrHints non_cacheable;
    non_cacheable.attrs = MemAttr_NC;

    MemoryAttributes nc_memattrs;
    nc_memattrs.memtype      = MemType_Normal;
    nc_memattrs.outer        = non_cacheable;
    nc_memattrs.inner        = non_cacheable;
    nc_memattrs.shareability = Shareability_OSH;

    return nc_memattrs;
```

G1.2.5.13 S1ConstrainUnpredictableRESMAIR

```
// S1ConstrainUnpredictableRESMAIR()
// =================================
// Determine whether a reserved value occupies MAIR_ELx.AttrN

boolean S1ConstrainUnpredictableRESMAIR(bits(8) attr, boolean s1aarch64)
    case attr of
        when '0000xxxx' return attr<1:0> != '00';
        when 'xxxx0000' return TRUE;
        otherwise       return FALSE;
```

G1.2.5.14 S1DecodeMemAttrs

```
// S1DecodeMemAttrs()
// ==================
// Decode MAIR-format memory attributes assigned in stage 1

MemoryAttributes S1DecodeMemAttrs(bits(8) attr_in, bits(2) sh, boolean s1aarch64)
    bits(8) attr = attr_in;
    if S1ConstrainUnpredictableRESMAIR(attr, s1aarch64) then
        // Map reserved encodings to an allocated encoding
        (-, attr) = ConstrainUnpredictableBits(Unpredictable_RESMAIR, 8);

    MemoryAttributes memattrs;
    case attr of
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when '0000xxxx' // Device memory
            mematTrs.memtype = MemType_Device;
            mematTrs.device  = DecodeDevice(attr<3:2>);
        otherwise
            mematTrs.memtype = MemType_Normal;
            mematTrs.outer   = DecodeLDFAttr(attr<7:4>);
            mematTrs.inner   = DecodeLDFAttr(attr<3:0>);

    mematTrs.shareability = DecodeShareability(sh);

    return mematTrs;

// S1DecodeMemAttrs()
// ==================
// Decode MAIR-format memory attributes assigned in stage 1

MemoryAttributes S1DecodeMemAttrs(bits(8) attr_in, bits(2) sh, boolean s1aarch64,
                                  S1TTWParams walkparams, AccessType acctype)
    MemoryAttributes mematTrs = S1DecodeMemAttrs(attr_in, sh, s1aarch64);

    return mematTrs;
```

G1.2.5.15 S2CombineS1AttrHints

```
// S2CombineS1AttrHints()
// ======================
// Determine resultant Normal memory cacheability and allocation hints from
// combining stage 1 Normal memory attributes and stage 2 cacheability attributes.

MemAttrHints S2CombineS1AttrHints(MemAttrHints s1_attrhints, MemAttrHints s2_attrhints)
    MemAttrHints attrhints;

    if s1_attrhints.attrs == MemAttr_NC || s2_attrhints.attrs == MemAttr_NC then
        attrhints.attrs = MemAttr_NC;
    elsif s1_attrhints.attrs == MemAttr_WT || s2_attrhints.attrs == MemAttr_WT then
        attrhints.attrs = MemAttr_WT;
    else
        attrhints.attrs = MemAttr_WB;

    // Stage 2 does not assign any allocation hints
    // Instead, they are inherited from stage 1
    if attrhints.attrs != MemAttr_NC then
        attrhints.hints     = s1_attrhints.hints;
        attrhints.transient = s1_attrhints.transient;

    return attrhints;
```

G1.2.5.16 S2CombineS1Device

```
// S2CombineS1Device()
// ===================
// Determine resultant Device type from combining output memory attributes
// in stage 1 and Device attributes in stage 2

DeviceType S2CombineS1Device(DeviceType s1_device, DeviceType s2_device)
    if s1_device == DeviceType_nGnRnE || s2_device == DeviceType_nGnRnE then
        return DeviceType_nGnRnE;
    elsif s1_device == DeviceType_nGnRE || s2_device == DeviceType_nGnRE then
        return DeviceType_nGnRE;
    elsif s1_device == DeviceType_nGRE || s2_device == DeviceType_nGRE then
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        return DeviceType_nGRE;
    else
        return DeviceType_GRE;
```

G1.2.5.17 S2CombineS1MemAttrs

```
// S2CombineS1MemAttrs()
// =====================
// Combine stage 2 with stage 1 memory attributes

MemoryAttributes S2CombineS1MemAttrs(MemoryAttributes s1_mematTrs, MemoryAttributes s2_mematTrs,
                                     boolean s2aarch64)
    MemoryAttributes mematTrs;

    if s1_mematTrs.memtype == MemType_Device && s2_mematTrs.memtype == MemType_Device then
        mematTrs.memtype = MemType_Device;
        mematTrs.device  = S2CombineS1Device(s1_mematTrs.device, s2_mematTrs.device);
    elsif s1_mematTrs.memtype == MemType_Device then    // S2 Normal, S1 Device
        mematTrs = s1_mematTrs;
    elsif s2_mematTrs.memtype == MemType_Device then    // S2 Device, S1 Normal
        mematTrs = s2_mematTrs;
    else                                                // S2 Normal, S1 Normal
        mematTrs.memtype = MemType_Normal;
        mematTrs.inner   = S2CombineS1AttrHints(s1_mematTrs.inner, s2_mematTrs.inner);
        mematTrs.outer   = S2CombineS1AttrHints(s1_mematTrs.outer, s2_mematTrs.outer);

    mematTrs.shareability = S2CombineS1Shareability(s1_mematTrs.shareability,
                                                    s2_mematTrs.shareability);

    mematTrs.shareability = EffectiveShareability(mematTrs);
    return mematTrs;
```

G1.2.5.18 S2CombineS1Shareability

```
// S2CombineS1Shareability()
// =========================
// Combine stage 2 shareability with stage 1

Shareability S2CombineS1Shareability(Shareability s1_shareability,
                                     Shareability s2_shareability)

    if (s1_shareability == Shareability_OSH ||
            s2_shareability == Shareability_OSH) then
        return Shareability_OSH;
    elsif (s1_shareability == Shareability_ISH ||
            s2_shareability == Shareability_ISH) then
        return Shareability_ISH;
    else
        return Shareability_NSH;
```

G1.2.5.19 WalkMemAttrs

```
// WalkMemAttrs()
// ==============
// Retrieve memory attributes of translation table walk

MemoryAttributes WalkMemAttrs(bits(2) sh, bits(2) irgn, bits(2) orgn)
    MemoryAttributes walkmemattrs;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    walkmemattrs.memtype      = MemType_Normal;
    walkmemattrs.shareability = DecodeShareability(sh);
    walkmemattrs.inner        = DecodeSDFAttr(irgn);
    walkmemattrs.outer        = DecodeSDFAttr(orgn);

    return walkmemattrs;
```

G1.2.5.20 AlignmentFault

```
// AlignmentFault()
// ================
// Return a fault record indicating an Alignment fault not due to memory type has occured
// for a specific access

FaultRecord AlignmentFault(AccessDescriptor accdesc, bits(64) vaddress)
    FaultRecord fault = NoFault(accdesc, vaddress);
    fault.statuscode  = Fault_Alignment;

    return fault;
```

G1.2.5.21 ExclusiveFault

```
// ExclusiveFault()
// ================
// Return a fault record indicating a fault for an unsupported Exclusive access

FaultRecord ExclusiveFault(AccessDescriptor accdesc, bits(64) vaddress)
    FaultRecord fault = NoFault(accdesc, vaddress);
    fault.statuscode  = Fault_Exclusive;

    return fault;
```

G1.2.5.22 NoFault

```
// NoFault()
// =========
// Return a clear fault record indicating no faults have occured

FaultRecord NoFault()
    FaultRecord fault;

    fault.vaddress = bits(64) UNKNOWN;
    fault.statuscode  = Fault_None;
    fault.accessdesc  = AccessDescriptor UNKNOWN;
    fault.secondstage = FALSE;
    fault.s2fs1walk   = FALSE;

    return fault;

// NoFault()
// =========
// Return a clear fault record indicating no faults have occured for a specific access

FaultRecord NoFault(AccessDescriptor accdesc)
    FaultRecord fault;

    fault.statuscode  = Fault_None;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    fault.accessdesc  = accdesc;
    fault.secondstage = FALSE;
    fault.s2fs1walk   = FALSE;
    fault.write       = !accdesc.read && accdesc.write;

    return fault;

// NoFault()
// =========

FaultRecord NoFault(AccessDescriptor accdesc, bits(64) vaddress)
    FaultRecord fault = NoFault();
    fault.accessdesc  = accdesc;
    fault.write       = !accdesc.read && accdesc.write;
    fault.vaddress    = vaddress;

    return fault;
```

G1.2.5.23 S1TranslationRegime

```
// S1TranslationRegime()
// =====================
// Stage 1 translation regime for the given Exception level

bits(2) S1TranslationRegime(bits(2) el)
    if el != EL0 then
        return el;

    else
        return EL1;

// S1TranslationRegime()
// =====================
// Returns the Exception level controlling the current Stage 1 translation regime. For the most
// part this is unused in code because the System register accessors (SCTLR_ELx, etc.) implicitly
// return the correct value.

bits(2) S1TranslationRegime()
    return S1TranslationRegime(PSTATE.EL);
```

G1.2.5.24 AddressDescriptor

```
constant integer FINAL_LEVEL = 3;

// AddressDescriptor
// =================
// Descriptor used to access the underlying memory array.

type AddressDescriptor is (
    FaultRecord      fault,      // fault.statuscode indicates whether the address is valid
    MemoryAttributes memattrs,
    FullAddress      paddress,
    bits(64)         vaddress
)
```

G1.2.5.25 ContiguousSize

```
// ContiguousSize()
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// ================
// Return the number of entries log 2 marking a contiguous output range

integer ContiguousSize(TGx tgx, integer level)
    case tgx of
        when TGx_4KB
            assert level IN {1, 2, 3};
            return 4;
        when TGx_16KB
            assert level IN {2, 3};
            return if level == 2 then 5 else 7;
        when TGx_64KB
            assert level IN {2, 3};
            return 5;
```

G1.2.5.26 CreateAddressDescriptor

```
// CreateAddressDescriptor()
// =========================
// Set internal members for address descriptor type to valid values

AddressDescriptor CreateAddressDescriptor(FullAddress pa, MemoryAttributes memattrs,
                                          AccessDescriptor accdesc)
    AddressDescriptor addrdesc;

    addrdesc.paddress = pa;
    addrdesc.memattrs = memattrs;
    addrdesc.fault    = NoFault(accdesc);

    return addrdesc;

// CreateAddressDescriptor()
// =========================
// Set internal members for address descriptor type to valid values

AddressDescriptor CreateAddressDescriptor(bits(64) va, FullAddress pa,
                                          MemoryAttributes memattrs, AccessDescriptor accdesc)
    AddressDescriptor addrdesc;

    addrdesc.paddress = pa;
    addrdesc.vaddress = va;
    addrdesc.memattrs = memattrs;
    addrdesc.fault    = NoFault(accdesc);

    return addrdesc;
```

G1.2.5.27 CreateFaultyAddressDescriptor

```
// CreateFaultyAddressDescriptor()
// ===============================
// Set internal members for address descriptor type with values indicating error

AddressDescriptor CreateFaultyAddressDescriptor(FaultRecord fault)
    AddressDescriptor addrdesc;

    addrdesc.vaddress = fault.vaddress;
    addrdesc.fault    = fault;

    return addrdesc;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.5.28 DescriptorType

```
// DescriptorType
// ==============
// Translation table descriptor formats

enumeration DescriptorType {
    DescriptorType_Table,
    DescriptorType_Leaf,
    DescriptorType_Invalid
};
```

G1.2.5.29 FetchDescriptor

```
// FetchDescriptor()
// =================
// Fetch a translation table descriptor

(FaultRecord, bits(N)) FetchDescriptor(bit ee, AddressDescriptor walkaddress,
                                       AccessDescriptor walkaccess, FaultRecord fault_in,
                                       integer N)
    // 64-bit descriptors for AArch64
    assert N == 64;
    bits(N) descriptor;
    FaultRecord fault = fault_in;

    PhysMemRetStatus memstatus;
    (memstatus, descriptor) = PhysMemRead(walkaddress, N DIV 8, walkaccess);
    if IsFault(memstatus) then
        constant boolean iswrite = FALSE;
        fault = HandleExternalTTWAbort(memstatus, iswrite, walkaddress,
                                       walkaccess, N DIV 8, fault);
        if IsFault(fault.statuscode) then
            return (fault, bits(N) UNKNOWN);

    if ee == '1' then
        descriptor = BigEndianReverse(descriptor);

    return (fault, descriptor);
```

G1.2.5.30 HasUnprivileged

```
// HasUnprivileged()
// =================
// Returns whether a translation regime serves EL0 as well as a higher EL

boolean HasUnprivileged(Regime regime)
    return (regime IN {
        Regime_EL10
    });
```

G1.2.5.31 Regime

```
// Regime
// ======
// Translation regimes
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
enumeration Regime {
    Regime_EL2,             // EL2
    Regime_EL10             // EL1&0
};
```

G1.2.5.32 RegimeUsingAArch32

```
// RegimeUsingAArch32()
// ====================
// Determine if the EL controlling the regime executes in AArch32 state

boolean RegimeUsingAArch32(Regime regime)
    return FALSE;
```

G1.2.5.33 S1TTWParams

```
// S1TTWParams
// ===========
// Register fields corresponding to stage 1 translation
// For A32-VMSA, if noted, they correspond to A32-LPAE (Long descriptor format)

type S1TTWParams is (
// A64-VMSA exclusive parameters
    bit         ha,         // TCR_ELx.HA
    bit         hd,         // TCR_ELx.HD
    bit         tbi,        // TCR_ELx.TBI{x}
    bit         tbid,       // TCR_ELx.TBID{x}
    bit         e0pd,       // TCR_EL1.E0PDx or TCR_EL2.E0PDx when HCR_EL2.E2H == '1'
    bits(3)     ps,         // TCR_ELx.{I}PS
    bits(6)     txsz,       // TCR_ELx.TxSZ

// Parameters common to both A64-VMSA & A32-VMSA (A64/A32)
    TGx         tgx,        // TCR_ELx.TGx      / Always TGx_4KB
    bits(2)     irgn,       // TCR_ELx.IRGNx    / TTBCR.IRGNx or HTCR.IRGN0
    bits(2)     orgn,       // TCR_ELx.ORGNx    / TTBCR.ORGNx or HTCR.ORGN0
    bits(2)     sh,         // TCR_ELx.SHx      / TTBCR.SHx or HTCR.SH0
    bit         hpd,        // TCR_ELx.HPD{x}   / TTBCR2.HPDx or HTCR.HPD
    bit         ee,         // SCTLR_ELx.EE     / SCTLR.EE or HSCTLR.EE
    bit         wxn,        // SCTLR_ELx.WXN    / SCTLR.WXN or HSCTLR.WXN
    bit         dc,         // HCR_EL2.DC       / HCR.DC
    MAIRType    mair        // MAIR_ELx         / MAIR1:MAIR0 or HMAIR1:HMAIR0
)
```

G1.2.5.34 SecurityStateForRegime

```
// SecurityStateForRegime()
// ========================
// Return the Security State of the given translation regime

SecurityState SecurityStateForRegime(Regime regime)
    case regime of
        when Regime_EL2     return SecurityStateAtEL(EL2);
        when Regime_EL10    return SecurityStateAtEL(EL1);
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.5.35 StageOA

```
// StageOA()
// =========
// Given the final walk state (a page or block descriptor), map the untranslated
// input address bits to the output address

FullAddress StageOA(bits(64) ia, TGx tgx, TTWState walkstate)
    // Output Address
    FullAddress oa;
    constant integer tsize = TranslationSize(tgx, walkstate.level);
    constant integer csize = (if walkstate.contiguous == '1' then
                                  ContiguousSize(tgx, walkstate.level)
                              else 0);
    constant AddressSize ia_msb = tsize + csize;
    oa.paspace = walkstate.baseaddress.paspace;
    oa.address = walkstate.baseaddress.address<51:ia_msb>:ia<ia_msb-1:0>;

    return oa;
```

G1.2.5.36 TGx

```
// TGx
// ===
// Translation granules sizes

enumeration TGx {
    TGx_4KB,
    TGx_16KB,
    TGx_64KB
};
```

G1.2.5.37 TGxGranuleBits

```
// TGxGranuleBits()
// ================
// Retrieve the address size, in bits, of a granule

AddressSize TGxGranuleBits(TGx tgx)
    case tgx of
        when TGx_4KB  return 12;
        when TGx_16KB return 14;
        when TGx_64KB return 16;
```

G1.2.5.38 TLBContext

```
// TLBContext
// ==========
// Translation context compared on TLB lookups and invalidations, promoting a TLB hit on match

type TLBContext is (
    SecurityState ss,
    Regime        regime,
    bits(16)      vmid,
    bits(16)      asid,
    bit           nG,
    PASpace       ipaspace, // Used in stage 2 lookups & invalidations only
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    boolean       includes_s1,
    boolean       includes_s2,
    boolean       use_vmid,
    bits(64)      ia,       // Input Address
    TGx           tg,
    bit           cnp,
    integer       level,    // Assist TLBI level hints (FEAT_TTL)
)
```

G1.2.5.39 TLBRecord

```
// TLBRecord
// =========
// Translation output as a TLB payload

type TLBRecord is (
    TLBContext    context,
    TTWState      walkstate,
    AddressSize   blocksize, // Number of bits directly mapped from IA to OA
    integer       contigsize,// Number of entries log 2 marking a contiguous output range
    bits(64)    s1descriptor, // Stage 1 leaf descriptor in memory (valid if the TLB caches stage 1)
    bits(64)    s2descriptor  // Stage 2 leaf descriptor in memory (valid if the TLB caches stage 2)
)
```

G1.2.5.40 TTWState

```
// TTWState
// ========
// Translation table walk state

type TTWState is (
    boolean             istable,
    integer             level,
    FullAddress         baseaddress,
    bit                 contiguous,
    bit                 nG,
    MemoryAttributes    memattrs,
    Permissions         permissions
)
```

G1.2.5.41 TranslationRegime

```
// TranslationRegime()
// ===================
// Select the translation regime given the target EL and PE state

Regime TranslationRegime(bits(2) el)
    if el == EL2 then
        return Regime_EL2;
    elsif el == EL1 then
        return Regime_EL10;
    elsif el == EL0 then
        return Regime_EL10;
    else
        Unreachable();
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

G1.2.5.42 TranslationSize

```
// TranslationSize()
// =================
// Compute the number of bits directly mapped from the input address
// to the output address

AddressSize TranslationSize(TGx tgx, integer level)
    granulebits = TGxGranuleBits(tgx);
    descsizelog2 = 3;
    blockbits   = (FINAL_LEVEL - level) * (granulebits - descsizelog2);

    return granulebits + blockbits;
```

G1.2.5.43 UseASID

```
// UseASID()
// =========
// Determine whether the translation context for the access requires ASID or is a global entry

boolean UseASID(TLBContext accesscontext)
    return HasUnprivileged(accesscontext.regime);
```

G1.2.5.44 UseVMID

```
// UseVMID()
// =========
// Determine whether the translation context for the access requires VMID to match a TLB entry

boolean UseVMID(Regime regime)
    return regime == Regime_EL10 && EL2Enabled();
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Part H
Appendixes

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Appendix H1
Architectural Constraints on UNPREDICTABLE Behaviors

This chapter describes the architectural constraints on UNPREDICTABLE behaviors in the Armv8 and later architectures. It contains the following sections:

- AArch64 CONSTRAINED UNPREDICTABLE behaviors.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H1.1 AArch64 CONSTRAINED UNPREDICTABLE behaviors

It contains the following sections:

- Overview of the constraints on AArch64 UNPREDICTABLE behaviors.
- SBZ or SBO fields in A64 instructions.
- CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values.
- CONSTRAINED UNPREDICTABLE behavior due to inadequate context synchronization.
- Translation table base address alignment.
- The Performance Monitors Extension.

- Syndrome register handling for CONSTRAINED UNPREDICTABLE instructions treated as UNDEFINED.
- Out of range virtual address.
- Mapping of non-idempotent memory locations using the Normal memory type.
- Instruction fetches from Device memory.
- Programming the CSSELR_EL1.{Level, InD, TnD} for a cache level that is not implemented.
- Crossing a page boundary with different memory types or Shareability attributes.
- Crossing a peripheral boundary with a Device access.
- CONSTRAINED UNPREDICTABLE behaviors with Load-Exclusive/Store-Exclusive pairs.
- CONSTRAINED UNPREDICTABLE behavior for A64 instructions.
- Out of range values of the Set/Way/Index fields in cache maintenance instructions.
- Reserved values in System and memory-mapped registers and translation table entries.
- CONSTRAINED UNPREDICTABLE behavior in Debug state.

H1.1.1 Overview of the constraints on AArch64 UNPREDICTABLE behaviors

The term UNPREDICTABLE describes a number of cases where the architecture has a feature that software must not use. For execution in AArch64 state, the Armv8-A and later architectures specify a narrow range of permitted behaviors. This range is the range of CONSTRAINED UNPREDICTABLE behavior. All implementations that are compliant with the architecture must follow the CONSTRAINED UNPREDICTABLE behavior.

Note

Software designed to be compatible with the Armv8-A and later architectures must not rely on these CONSTRAINED UNPREDICTABLE cases being handled in any way other than those listed under the heading CONSTRAINED UNPREDICTABLE.

H1.1.2 SBZ or SBO fields in A64 instructions

Some A64 instructions have (0) or (1) in the instruction decode to indicate *Should-Be-Zero*, SBZ, or *Should-Be-One*, SBO, as described in Fixed values in AArch64 instruction and System register descriptions. Except for specific cases identified in CONSTRAINED UNPREDICTABLE behaviors with Load-Exclusive/Store-Exclusive pairs, if the instruction bit pattern of an instruction is executed with these fields not having the *should be* values, one of the following must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction operates as if the bit had the *should-be* value.
- Any destination registers of the instruction become UNKNOWN.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H1.1.3 CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values

The Arm architecture allows copies of control values or data values to be cached in a cache or TLB. This can lead to UNPREDICTABLE behavior if the cache or TLB has not been correctly invalidated following a change of the control or data values.

Unless explicitly stated otherwise, the behavior of the PE is consistent with one of:

- The old data or control value.
- The new data or control value.
- An amalgamation of the old and new data or control values.

In an implementation that includes FEAT_TTCNP, this CONSTRAINED UNPREDICTABLE case can arise from misprogramming when setting TTBR.CnP to 1, as identified in the descriptions of the TTBR.CnP field. In this case, for a particular TTBR, the behavior of the PE is consistent with one of:

- The value of the translation table entry pointed to by that TTBR on one of the PEs within the Inner Shareable domain for which both the value of TTBR.CnP is 1 and the other conditions for sharing translation table entries pointed to by that TTBR are met.
- An amalgamation of the values of the translation table entries pointed to by that TTBR on two or more of the PEs within the Inner Shareable domain for which both the value of TTBR.CnP is 1 and the other conditions for sharing translation table entries pointed to by that TTBR are met.

Note

If the Effective value of a control or data value that determines the behavior of the PE results from the amalgamation of two or more values, then that Effective value must not generate a privilege violation. So, for example:

- Where the CONSTRAINED UNPREDICTABLE behavior occurs because inadequate invalidation of the TLB causes multiple hits in the TLB, the failure to invalidate the TLB by software executing at a given Exception level and Security state must not make it possible to access regions of memory with permissions or attributes that could not be accessed at that Exception level and Security state.
- Where the CONSTRAINED UNPREDICTABLE behavior occurs because of a programming error, on one or more PEs in the Inner Shareable domain, when using a TTBR.CnP value of 1 to share translation table entries, the misprogramming must not make it possible to access regions of memory with permissions or attributes that could not be accessed at the Exception level of that TTBR and the Security state corresponding to the translation table entries being shared.

Alternatively to this CONSTRAINED UNPREDICTABLE behavior, an implementation detecting multiple hits in a TLB might generate an exception, reporting the exception using the TLB conflict fault code, see TLB conflict abort.

The choice between the behaviors might, in some implementations, vary for each use of a control or data value.

H1.1.4 CONSTRAINED UNPREDICTABLE behavior due to inadequate context synchronization

The Arm architecture requires that changes to System registers must be synchronized before they take effect. This can lead to UNPREDICTABLE behavior if the synchronization has not been performed.

In these cases, the behavior of the PE is consistent with the unsynchronized control value being either the old value or the new value.

Where multiple control values are updated but not yet synchronized, each control value might independently be the old value or the new value.

In addition, where the unsynchronized control value applies to different areas of functionality, or what an implementation has constructed as different areas of functionality, those areas might independently treat the control value as being either the old value or the new value.

The choice between these behaviors might, in some implementations, vary for each use of a control value.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H1.1.5 Translation table base address alignment

In the translation table base registers TTBR0_EL1, TTBR1_EL1, TTBR0_EL2, register bits[48:*x*] hold the translation table base address, where *x* depends on the translation table granule size and the size of the addressed translation table, as described in Translation granules. Register bits[(*x*-1):0], unless redefined for another purpose, correspond to bits[(*x*-1):0] of the translation table base address and therefore are RES0.

Note

- When FEAT_LPA is implemented and the 64KB granule size is used, register bits[5:2] are redefined to hold bits[51:48] of the translation table base address.
- When FEAT_TTCNP is implemented register bit[0] is redefined as the CnP bit.

For these registers, if one or more RES0 bits in register bits [(x-1):0] does not have a value of 0, this can result in a misaligned translation table base address. In this case, one of the following behaviors must occur:

- The field that is defined to be RES0 is treated as if all the bits had a value of 0:
  - The value read back might be the value written or it might be zero.
- The calculation of an address for a translation table walk using those registers might be corrupted in those bits that are nonzero.

For more information, see the appropriate TTBR.BADDR field description.

H1.1.6 The Performance Monitors Extension

The following subsections describe CONSTRAINED UNPREDICTABLE behaviors when accessing the Performance Monitors Extension in AArch64 state:

- CONSTRAINED UNPREDICTABLE accesses to PMXEVTYPER_EL0 or PMXEVETYPER_EL0.
- CONSTRAINED UNPREDICTABLE accesses to PMEVCNTR<n>_EL0 and PMEVTYPER<n>_EL0.
- CONSTRAINED UNPREDICTABLE behavior caused by MDCR_EL2.HPMN.

H1.1.6.1 CONSTRAINED UNPREDICTABLE accesses to PMXEVTYPER_EL0 or PMXEVETYPER_EL0

If PMSELR_EL0.SEL is greater than the number of event counters accessible at this Exception level, accesses to PMXEVTYPER_EL0 and PMXEVCNTR_EL0 can cause CONSTRAINED UNPREDICTABLE behavior. This occurs when one of the following is true:

- If PMSELR_EL0.SEL is not equal to 31, and PMSELR_EL0.SEL is greater than or equal to PMCR_EL0.N, and the PE is executing in EL2 or EL3.
- If FEAT_SEL2 is disabled or is not implemented, PMSELR_EL0.SEL is not 31, and PMSELR_EL0.SEL is greater than or equal to PMCR_EL0.N, and the PE is executing in Secure EL1 or Secure EL0.
- If PMSELR_EL0.SEL is not 31, and PMSELR_EL0.SEL is greater than or equal to MDCR_EL2.HPMN, and the PE is executing in EL0 or EL1.

In these cases, one of the following behaviors must occur:

- Accesses to PMXEVTYPER_EL0 or PMXEVCNTR_EL0 from that state are UNDEFINED.
- Accesses to PMXEVTYPER_EL0 or PMXEVCNTR_EL0 from that state behave as RAZ/WI.
- Accesses to PMXEVTYPER_EL0 or PMXEVCNTR_EL0 from that state execute as NOPs.
- Accesses to PMXEVTYPER_EL0 or PMXEVCNTR_EL0 from that state behave as if PMSELR_EL0.SEL contains an UNKNOWN value that is less than the number of counters accessible at the current Exception level and Security state.
- Accesses to PMXEVTYPER_EL0 or PMXEVCNTR_EL0 from that state behave as if PMSELR_EL0.SEL is 31.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If EL2 is implemented and enabled in the current Security state, and PMSELR_EL0.SEL is less than the number of accessible event counters but greater than or equal to the number of accessible counters at this Exception level, access to PMXEVTYPER_EL0 or PMXEVCNTR_EL0 from EL1 or a permitted access from EL0 is trapped to EL2.

If PMSELR_EL0.SEL is equal to 31, then one of the following behaviors must occur:

- Accesses to PMXEVCNTR_EL0 are UNDEFINED.
- Accesses to PMXEVCNTR_EL0 behave as RAZ/WI.
- Accesses to PMXEVCNTR_EL0 execute as NOPs.
- Accesses to PMXEVCNTR_EL0 behave as if PMSELR_EL0.SEL contains an unknown value that is less than the number of counters accessible at the current Exception level and Security state.

H1.1.6.2 CONSTRAINED UNPREDICTABLE accesses to PMEVCNTR<n>_EL0 and PMEVTYPER<n>_EL0

If <n> is greater than the number of event counters available in the current Exception level and state, reads and writes of PMEVCNTR<n>_EL0 and PMEVTYPER<n>_EL0 are CONSTRAINED UNPREDICTABLE, the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP.
- If EL2 is implemented and enabled in the current Security state, for an access to PMEVCNTR<n>_EL0 or PMEVTYPER<n>_EL0 from EL1 or a permitted access from EL0, if the counter is implemented but not accessible at the current Exception level, the register access is trapped to EL2.

Accesses from EL0 to PMEVCNTR<n>_EL0 are permitted when EL1 is using AArch64 and the value of PMUSERENR_EL0.{ER, EN} are both 1.

Accesses from EL0 to PMEVTYPER<n>_EL0 are permitted when EL1 is using AArch64 and the value of PMUSERENR_EL0.EN is 1.

H1.1.6.3 CONSTRAINED UNPREDICTABLE behavior caused by MDCR_EL2.HPMN

If PMCR_EL0.N is nonzero, and one of the following is true:

- FEAT_HPMN0 is not implemented and MDCR_EL2.HPMN is set to 0.
- MDCR_EL2.HPMN is set to a value greater than PMCR_EL0.N.

The following CONSTRAINED UNPREDICTABLE behavior applies:

- The value returned by a direct read of MDCR_EL2.HPMN is UNKNOWN.
- Either:
  - An UNKNOWN number of counters are reserved for EL2 use. That is, the PE behaves as if MDCR_EL2.HPMN is set to an UNKNOWN nonzero value less than PMCR_EL0.N.
  - All counters are reserved for EL2 and EL3 use, meaning no counters are accessible from EL1 and EL0.

H1.1.7 Syndrome register handling for CONSTRAINED UNPREDICTABLE instructions treated as UNDEFINED

When a CONSTRAINED UNPREDICTABLE instruction is treated as UNDEFINED, ESR_ELx is UNKNOWN.

Note

The value written to ESR_ELx must be consistent with a value that could be created as the result of an exception from the same Exception level that generated the exception, but was the result of a situation that is not CONSTRAINED UNPREDICTABLE at that Exception level. This is to avoid a possible privilege violation.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H1.1.8 Out of range virtual address

If the PE executes a load or store instruction with tagged addressing disabled in the current translation regime, and where the computed virtual address, total access size, and alignment mean that it accesses the bytes at `0xFFFF_FFFF_FFFF_FFFF` and `0x0000_0000_0000_0000`, then the bytes that appear to be from `0x0000_0000_0000_0000` onwards are accessed at an UNKNOWN address. It is permitted for the UNKNOWN address used to be a fixed value such that it always generates an MMU fault.

If the PE executes a load or store instruction with tagged addressing enabled in the current translation regime, and where the computed address, total access size, and alignment mean that it accesses the bytes at `0xFFFF_FFFF_FFFF_FFFF` and `0x0000_0000_0000_0000`, then the bytes that appear to be from `0x0000_0000_0000_0000` onwards are accessed at an UNKNOWN address and the tags associated with address also become UNKNOWN. It is permitted for the UNKNOWN address used to be a fixed value such that it always generates an MMU fault.

Note

Because of Program Counter alignment constraints, it is impossible for a PE to fetch an A64 instruction that includes both the byte at virtual address `0xFFFF_FFFF_FFFF_FFFF` and the byte at virtual address `0x0000_0000_0000_0000`.

H1.1.9 Mapping of non-idempotent memory locations using the Normal memory type

If non-idempotent memory locations are mapped using the Normal memory type, the state of the non-idempotent memory location may become corrupted in following circumstances:

- Speculative read accesses may cause accesses to the non-idempotent memory locations that would not occur as part of a simple sequential execution.
- Writes to non-idempotent memory locations might be merged or split. In this case, the number and size of writes seen by the memory location might not be the number and size that occur as part of a simple sequential execution.

H1.1.10 Instruction fetches from Device memory

Instruction fetches from Device memory are CONSTRAINED UNPREDICTABLE.

If a location in memory has the Device attribute and is not marked as execute-never, then an implementation might perform speculative instruction accesses to this memory location at times when address translation is enabled.

For an instruction fetch from a memory location with the Device attribute that is not marked as execute-never for the current Exception level, an implementation can either:

- Treat the instruction fetch as if it were to a memory location with the Normal, Non-cacheable attribute.
- Take a Permission fault.

H1.1.11 Programming the CSSELR_EL1.{Level, InD, TnD} for a cache level that is not implemented

If the CSSELR_EL1.{Level, InD, TnD} is programmed to a cache level that is not implemented, then a read of CSSELR_EL1 returns an UNKNOWN value in CSSELR_EL1.{Level, InD, TnD}.

If CSSELR_EL1.{Level, InD, TnD} is programmed to a cache level that is not implemented, then on a read of CCSIDR_EL1 an implementation must perform one of the following behaviors:

- The CCSIDR_EL1 read is treated as a NOP.
- The CCSIDR_EL1 read is UNDEFINED.
- The CCSIDR_EL1 read returns an UNKNOWN value.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_CCIDX is implemented, CCSIDR2_EL1 is implemented. If CSSELR_EL1.{Level, InD, TnD} is programmed to a cache level that is not implemented, then on a read of CCSIDR2_EL1 an implementation must perform one of the following behaviors:

- The CCSIDR2_EL1 read is treated as a NOP.
- The CCSIDR2_EL1 read is UNDEFINED.
- The CCSIDR2_EL1 read returns an UNKNOWN value.

H1.1.12 Crossing a page boundary with different memory types or Shareability attributes

If a single load or store instruction generates multiple memory accesses, such that the total set of accesses crosses a page boundary to a memory location that has a different memory type, Normal or Device, or Shareability attribute results in CONSTRAINED UNPREDICTABLE behavior. In this case, the implementation must perform one of the following behaviors:

- Each memory access generated by the instruction uses the memory type and Shareability attribute associated with its own address.
- The instruction generates an Alignment fault caused by the memory type.
  For the EL1&0 translation regime, if the stage 1 translation generated the mismatch, the resulting exception is taken to EL1.
- The instruction executes as a NOP.

H1.1.13 Crossing a peripheral boundary with a Device access

Performing memory accesses from one load or store instruction to Device memory that crosses a boundary corresponding to the smallest translation granule size of the implementation causes CONSTRAINED UNPREDICTABLE behavior. In this case, the implementation performs one of the following behaviors:

- All memory accesses generated by the instruction are performed as if the boundary has no effect on the memory accesses.
- All memory accesses generated by the instruction are performed as if the boundary has no effect on the memory accesses except that there is no guarantee of ordering between memory accesses.
- The instruction generates an alignment fault caused by the memory type.
- The instruction generates an Alignment fault caused by the memory type.
  For the EL1&0 translation regime, if the stage 1 translation generated the mismatch, the resulting exception is taken to EL1.
- The instruction executes as a NOP.

Note

The boundary referred to is between two Device memory regions that are both:

- Of the size of the smallest implemented translation granule.
- Aligned to the size of the smallest implemented translation granule.

H1.1.14 CONSTRAINED UNPREDICTABLE behaviors with SVE memory accesses

The CONSTRAINED UNPREDICTABLE behaviors that are associated with memory accesses due to loads and stores also apply to SVE vector load and store instructions. The CONSTRAINED UNPREDICTABLE behaviors are described in Crossing a page

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

boundary with different memory types or Shareability attributes and Crossing a peripheral boundary with a Device access.

The CONSTRAINED UNPREDICTABLE behaviors apply to the following SVE memory accesses:

- When an SVE unpredicated contiguous load or store instruction accesses an address range that crosses a boundary between memory types, Shareability attributes, or peripherals.
- When an SVE predicated contiguous load or store instruction performs memory accesses that are associated with Active elements on both sides of a boundary between different memory types, Shareability attributes, or peripherals.
- When an SVE predicated non-contiguous load or store instruction performs a memory access that is associated with an Active element that crosses a boundary between memory types, Shareability attributes, or peripherals.

Memory addresses that are associated with *Inactive elements* cannot trigger CONSTRAINED UNPREDICTABLE behaviors.

If SVE vector loads and stores trigger a CONSTRAINED UNPREDICTABLE behavior that then generates an alignment fault, the fault is handled in the same as any other synchronous memory fault caused by an SVE load or store instruction.

H1.1.15 CONSTRAINED UNPREDICTABLE behaviors with Load-Exclusive/Store-Exclusive pairs

Load-Exclusive and Store-Exclusive instruction usage restrictions defines a Load-Exclusive/Store-Exclusive pair, and identifies various CONSTRAINED UNPREDICTABLE behaviors associated with using Load-Exclusive/Store-Exclusive pairs. In summary, these cases are:

- The target virtual address of a `StoreExcl` instruction is different from the virtual address of the preceding `LoadExcl` instruction in the same thread of execution.
- The transaction size of a `StoreExcl` instruction is different from the transaction size of the preceding `LoadExcl` instruction in the same thread of execution.
- The `StoreExcl` instruction accesses a different number of registers than the preceding `LoadExcl` instruction in the same thread of execution.
- The memory attributes for a `StoreExcl` instruction are different from the memory attributes for the preceding `LoadExcl` instruction in the same thread of execution, either:
  - Because the translation of the accessed address changes between the `LoadExcl` instruction and the `StoreExcl` instruction.
  - Because the `LoadExcl` instruction and the `StoreExcl` instruction use different virtual addresses, with different attributes, that point to the same physical address.

In addition, the effect of a data or unified cache invalidate, clean, or clean and invalidate instruction on a local or global Exclusives monitor that is in the Exclusive Access state is CONSTRAINED UNPREDICTABLE.

See the descriptions in Load-Exclusive and Store-Exclusive instruction usage restrictions for the permitted behavior in each of these cases, and any constraints that might apply to whether the case is CONSTRAINED UNPREDICTABLE.

H1.1.16 CONSTRAINED UNPREDICTABLE behavior for A64 instructions

This section lists the CONSTRAINED UNPREDICTABLE behavior for the different A64 instructions listed in A64 Base Instruction Descriptions and A64 Advanced SIMD and Floating-point Instruction Descriptions.

H1.1.16.1 LD64B

For a description of this instruction and encoding, see LD64B.

H1.1.16.1.1 CONSTRAINED UNPREDICTABLE behavior

If a memory location does not support this instruction, then one of the following behaviors must occur:

- A stage 1 Data Abort, reported using the DFSC code of 110101, is generated.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The instruction performs the memory accesses, but the accesses are not single-copy atomic above the byte level.

H1.1.16.2 LDAXP

For a description of this instruction and the encoding, see LDAXP.

H1.1.16.2.1 CONSTRAINED UNPREDICTABLE behavior

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs a load using the specified addressing mode, and the transfer register is set to an UNKNOWN value.

H1.1.16.3 LDNP

For a description of this instruction and the encoding, see LDNP.

H1.1.16.3.1 CONSTRAINED UNPREDICTABLE behavior

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs a load using the specified addressing mode, and the transfer register is set to an UNKNOWN value.

H1.1.16.4 LDNP (SIMD&FP)

For a description of this instruction and the encoding, see LDNP (SIMD&FP).

H1.1.16.4.1 CONSTRAINED UNPREDICTABLE behavior

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs a load using the specified addressing mode, and the transfer register is set to an UNKNOWN value.

H1.1.16.5 LDP and LDIAPP

For a description of these instructions and the encoding, see LDP and LDIAPP.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H1.1.16.5.1 CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `(t == n || t2 == n) && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs a load using the specified addressing mode, and the base register is set to an UNKNOWN value. In addition, if an exception occurs during such an instruction, the base register might be corrupted so that the instruction cannot be repeated.

If t == t2, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs all of the loads using the specified addressing mode, and the transfer register is set to an UNKNOWN value.

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

H1.1.16.6 LDP (SIMD&FP)

For a description of this instruction and the encoding, see LDP (SIMD&FP).

H1.1.16.6.1 CONSTRAINED UNPREDICTABLE behavior

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs a load using the specified addressing mode, and the transfer register is set to an UNKNOWN value.

H1.1.16.7 LDPSW

For a description of this instruction and the encoding, see LDPSW.

H1.1.16.7.1 CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `(t == n || t2 == n) && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs a load using the specified addressing mode, and the base register is set to an UNKNOWN value. In addition, if an exception occurs during such an instruction, the base register might be corrupted so that the instruction cannot be repeated.

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs all of the loads using the specified addressing mode, and the register loaded is set to an UNKNOWN value.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

H1.1.16.8 LDR (immediate)

For a description of this instruction and the encoding, see LDR (immediate).

H1.1.16.8.1 CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `n == t && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the load using the specified addressing mode, and the base register is set to an UNKNOWN value. In addition, if an exception occurs during such an instruction, the base register might be corrupted so that the instruction cannot be repeated.

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

H1.1.16.9 LDRB (immediate)

For a description of this instruction and the encoding, see LDRB (immediate).

H1.1.16.9.1 CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `n == t && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the load using the specified addressing mode, and the base register is set to an UNKNOWN value. In addition, if an exception occurs during such an instruction, the base register might be corrupted so that the instruction cannot be repeated.

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

H1.1.16.10 LDRH (immediate)

For a description of this instruction and the encoding, see LDRH (immediate).

H1.1.16.10.1 CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `n == t && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The instruction performs the load using the specified addressing mode, and the base register is set to an UNKNOWN value. In addition, if an exception occurs during such an instruction, the base register might be corrupted so that the instruction cannot be repeated.

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

H1.1.16.11 LDRSB (immediate)

For a description of this instruction and the encoding, see LDRSB (immediate).

H1.1.16.11.1 CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `n == t && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the load using the specified addressing mode, and the base register is set to an UNKNOWN value. In addition, if an exception occurs during such an instruction, the base register might be corrupted so that the instruction cannot be repeated.

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

H1.1.16.12 LDRSH (immediate)

For a description of this instruction and the encoding, see LDRSH (immediate).

H1.1.16.12.1 CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `n == t && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the load using the specified addressing mode, and the base register is set to an UNKNOWN value. In addition, if an exception occurs during such an instruction, the base register might be corrupted so that the instruction cannot be repeated.

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

H1.1.16.13 LDRSW (immediate)

For a description of this instruction and the encoding, see LDRSW (immediate).

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H1.1.16.13.1 CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `n == t && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the load using the specified addressing mode, and the base register is set to an UNKNOWN value. In addition, if an exception occurs during such an instruction, the base register might be corrupted so that the instruction cannot be repeated.

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

H1.1.16.14 LDXP

For a description of this instruction and the encoding, see LDXP.

H1.1.16.14.1 CONSTRAINED UNPREDICTABLE behavior

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs a load using the specified addressing mode, and the transfer register is set to an UNKNOWN value.

H1.1.16.15 ST64B

For a description of this instruction and encoding, see ST64B.

H1.1.16.15.1 CONSTRAINED UNPREDICTABLE behavior

If a memory location does not support this instruction, then one of the following behaviors must occur:

- A stage 1 Data Abort, reported using the DFSC code of 110101, is generated.
- The instruction performs the memory accesses, but the accesses are not single-copy atomic above the byte level.

H1.1.16.16 STP and STILP

For a description of these instructions and the encoding, see STP and STILP.

H1.1.16.16.1 CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `(t == n || t2 == n) && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs a store using the specified addressing mode but the value stored is UNKNOWN.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

H1.1.16.17 STLXP

For a description of this instruction and the encoding, see STLXP.

H1.1.16.17.1 CONSTRAINED UNPREDICTABLE behavior

If `s == t || (s == t2)`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to the specified address, but the value stored is UNKNOWN.

If `s == n && n != 31` then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to an UNKNOWN address.

H1.1.16.18 STLXR

For a description of this instruction and the encoding, see STLXR.

H1.1.16.18.1 CONSTRAINED UNPREDICTABLE behavior

If `s == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to the specified address, but the value stored is UNKNOWN.

If `s == n && n != 31` then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to an UNKNOWN address.

H1.1.16.19 STLXRB

For a description of this instruction and the encoding, see STLXRB.

H1.1.16.19.1 CONSTRAINED UNPREDICTABLE behavior

If `s == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to the specified address, but the value stored is UNKNOWN.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If `s == n && n != 31` then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to an UNKNOWN address.

H1.1.16.20 STLXRH

For a description of this instruction and the encoding, see STLXRH.

H1.1.16.20.1 CONSTRAINED UNPREDICTABLE behavior

If `s == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to the specified address, but the value stored is UNKNOWN.

If `s == n && n != 31` then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to an UNKNOWN address.

H1.1.16.21 STR (immediate)

For a description of this instruction and the encoding, see STR (immediate).

H1.1.16.21.1 CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `n == t && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs a store using the specified addressing mode but the value stored is UNKNOWN.

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

H1.1.16.22 STRB (immediate)

For a description of this instruction and the encoding, see STRB (immediate).

H1.1.16.22.1 CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `n == t && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs a store using the specified addressing mode but the value stored is UNKNOWN.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

H1.1.16.23 STRH (immediate)

For a description of this instruction and the encoding, see STRH (immediate).

H1.1.16.23.1 CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `n == t && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs a store using the specified addressing mode but the value stored is UNKNOWN.

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

H1.1.16.24 STXP

For a description of this instruction and the encoding, see STXP.

H1.1.16.24.1 CONSTRAINED UNPREDICTABLE behavior

If `s == t || (s == t2)`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to the specified address, but the value stored is UNKNOWN.

If `s == n && n != 31` then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to an UNKNOWN address.

H1.1.16.25 STXR

For a description of this instruction and the encoding, see STXR.

H1.1.16.25.1 CONSTRAINED UNPREDICTABLE behavior

If `s == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to the specified address, but the value stored is UNKNOWN.

If `s == n && n != 31` then one of the following behaviors must occur:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to an UNKNOWN address.

H1.1.16.26 STXRB

For a description of this instruction and the encoding, see STXRB.

H1.1.16.26.1 CONSTRAINED UNPREDICTABLE behavior

If `s == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to the specified address, but the value stored is UNKNOWN.

If `s == n && n != 31` then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to an UNKNOWN address.

H1.1.16.27 STXRH

For a description of this instruction and the encoding, see STXRH.

H1.1.16.27.1 CONSTRAINED UNPREDICTABLE behavior

If `s == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to the specified address, but the value stored is UNKNOWN.

If `s == n && n != 31` then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to an UNKNOWN address.

H1.1.17 Out of range values of the Set/Way/Index fields in cache maintenance instructions

In the cache maintenance by set/way instructions DC CISW, DC CSW, and DC ISW, if any set/way/index argument is larger than the value supported by the implementation, then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED.
- The instruction performs cache maintenance on one of:
  - No cache lines.
  - A single arbitrary cache line.
  - Multiple arbitrary cache lines.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H1.1.18 Reserved values in System and memory-mapped registers and translation table entries

Unless otherwise stated in this manual, all unallocated or reserved values of fields with allocated values within AArch64 System registers, memory-mapped registers, and translation table entries behave in one of the following ways:

- The unallocated value maps onto any of the allocated values, but otherwise does not cause CONSTRAINED UNPREDICTABLE behavior.
- The unallocated value causes effects that could be achieved by a combination of more than one of the allocated values.
- The unallocated value causes the field to have no functional effect.

Unless otherwise stated, when a direct write of a System register or memory-mapped register writes an unallocated value to a field, if that value is unallocated in all contexts then a subsequent read of the field returns an UNKNOWN value.

H1.1.19 CONSTRAINED UNPREDICTABLE behavior in Debug state

Behavior in Debug state of this manual describes the CONSTRAINED UNPREDICTABLE behaviors that are specifically associated with Debug state.

H1.1.20 PMSAv64 CONSTRAINED UNPREDICTABLE behavior

This section lists the CONSTRAINED UNPREDICTABLE behaviors of the MPUs in PMSAv64.

If HCR_EL2.VM is 1 and SCTLR_EL2.{M, BR} is {0, 0}, then the behavior is CONSTRAINED UNPREDICTABLE with the following permitted behaviors:

- The stage 2 memory attribute becomes UNKNOWN.
- Raises a stage 2 level 0 translation fault.

Any accesses to an MPU base address register PRBAR_ELx, limit address register PRLAR_ELx, or select register PSERLR_ELx above the number of implemented regions specified by MPUIR_ELx.REGION is CONSTRAINED UNPREDICTABLE. The CONSTAINED UNPREDICTABLE behavior is one of:

- Reads of base, limit or select registers for an unimplemented region returns an UNKNOWN value.
- Writes to base, limit or select registers for an unimplemented region causes those registers to become UNKNOWN.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Appendix H2
Recommendations for Reporting Memory Attributes on an Interconnect

This appendix describes the Arm recommendations for reporting the memory attributes that are assigned by the PE. It contains the following section:

- Arm recommendations for reporting memory attributes on an interconnect.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H2.1 Arm recommendations for reporting memory attributes on an interconnect

The Arm architecture defines the architectural interface between software and the PE hardware. This means the mechanisms by which different memory type and Cacheability attributes are presented on an interface to an interconnect fabric such as AMBA® AXI are, strictly, outside the scope of the architecture. This appendix describes an approach for the interface between a PE implementation and an interconnect fabric that Arm strongly recommends, but these recommendations do not form part of the Arm architecture.

H2.1.1 Effect of microarchitectural choices on memory attributes

Implementations of the Arm architecture permit considerable variability in the presentation of memory attributes on the interconnect fabric, particularly in cases where the PE implementation does not provide optimized support for a memory type. For example, an implementation might treat Write-Through locations as Non-cacheable at some level of cache, because functionally this is consistent with the definition of Write-Through, but for the particular implementation the performance trade-off does not merit the hardware directly providing Write-Through capability. However, in such implementations, the assigned memory attributes are not changed by the microarchitectural choices. The microarchitecture simply implements different ways of handling some memory attributes.

Therefore, Arm strongly recommended that where any or all of the following memory attributes are presented on the interface between a PE and an interconnect fabric, the attributes that are presented are completely consistent with the attributes defined by the translation system:

- The memory type, Normal or Device.
- The Early Write Acknowledgment attribute.
- The ordering requirements.
- The Shareability.
- The Cacheability, including where practicable, the allocation hints.

H2.1.1.1 Effect when memory accesses are forced to be Non-cacheable

Arm also strongly recommends that the effects of forcing accesses to Normal memory to be Non-cacheable, as described in Enabling and disabling the caching of memory accesses for AArch64, are reflected on the interconnect by the memory type and attributes used for memory transactions generated while the cache is disabled.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Appendix H3
Arm Pseudocode Definition

This appendix provides a definition of the pseudocode that is used in this manual, and defines some *helper* procedures and functions that are used by pseudocode. It contains the following sections:

- About the Arm pseudocode.
- Pseudocode for instruction descriptions.
- Data types.
- Operators.
- Statements and control structures.
- Built-in functions.
- Miscellaneous helper procedures and functions.
- Arm pseudocode definition index.

Note

This appendix is not a formal language definition for the pseudocode. It is a guide to help understand the use of Arm pseudocode. This appendix is not complete. Changes are planned for future releases.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H3.1 About the Arm pseudocode

The Arm pseudocode provides precise descriptions of some areas of the Arm architecture. This includes description of the decoding and operation of all valid instructions. Pseudocode for instruction descriptions gives general information about this instruction pseudocode, including its limitations.

The following sections describe the Arm pseudocode in detail:

- Data types.
- Operators.
- Statements and control structures.

Built-in functions and Miscellaneous helper procedures and functions describe some built-in functions and pseudocode helper functions that are used by the pseudocode functions that are described elsewhere in this manual. Arm pseudocode definition index contains the indexes to the pseudocode.

H3.1.1 General limitations of Arm pseudocode

The pseudocode statements `IMPLEMENTATION_DEFINED`, `SEE`, `UNDEFINED`, and `UNPREDICTABLE` indicate behavior that differs from that indicated by the pseudocode being executed. If one of them is encountered:

- Earlier behavior indicated by the pseudocode is only specified as occurring to the extent required to determine that the statement is executed.
- No subsequent behavior indicated by the pseudocode occurs.

For more information, see Special statements.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H3.2 Pseudocode for instruction descriptions

Each instruction description includes pseudocode that provides a precise description of what the instruction does, subject to the limitations described in General limitations of Arm pseudocode and Limitations of the instruction pseudocode.

In the instruction pseudocode, instruction fields are referred to by the names shown in the encoding diagram for the instruction. Instruction encoding diagrams and instruction pseudocode gives more information about the pseudocode provided for each instruction.

H3.2.1 Instruction encoding diagrams and instruction pseudocode

Instruction descriptions in this manual contain:

- An Encoding section, containing one or more encoding diagrams, each followed by some encoding-specific pseudocode that translates the fields of the encoding into inputs for the common pseudocode of the instruction, and picks out any encoding-specific special cases.
- An Operation section, containing common pseudocode that applies to all of the encodings being described. The Operation section pseudocode contains a call to the `EncodingSpecificOperations()` function, either at its start or only after a condition code check performed by `if ConditionPassed() then`.

An encoding diagram specifies each bit of the instruction as one of the following:

- An obligatory 0 or 1, represented in the diagram as 0 or 1. If this bit does not have this value, the encoding corresponds to a different instruction.

An encoding diagram matches an instruction if all obligatory bits are identical in the encoding diagram and the instruction.

In the context of the instruction pseudocode, the execution model for an instruction is:

1. Find all encoding diagrams that match the instruction. It is possible that no encoding diagram matches. In that case, abandon this execution model and consult the relevant instruction set chapter instead to find out how the instruction is to be treated. The bit pattern of such an instruction is usually reserved and UNDEFINED, though there are some other possibilities. For example, unallocated hint instructions are documented as being reserved and executed as NOPs.
2. If the operation pseudocode for the matching encoding diagrams starts with a condition code check, perform that check. If the condition code check fails, abandon this execution model and treat the instruction as a NOP. If there are multiple matching encoding diagrams, either all or none of their corresponding pieces of common pseudocode start with a condition code check.
3. Perform the encoding-specific pseudocode for each of the matching encoding diagrams independently and in parallel. Each such piece of encoding-specific pseudocode starts with a bitstring variable for each named bit or multi-bit field in its corresponding encoding diagram, named the same as the bit or multi-bit field and initialized with the values of the corresponding bit or bits from the bit pattern of the instruction.

   In a few cases, the encoding diagram contains more than one bit or field with same name. In these cases, the values of the different instances of those bits or fields must be identical. The encoding-specific pseudocode contains a special case using the `Consistent()` function to specify what happens if they are not identical. `Consistent()` returns `TRUE` if all instruction bits or fields with the same name as its argument have the same value, and `FALSE` otherwise.

   If there are multiple matching encoding diagrams, all but one of the corresponding pieces of pseudocode must contain a special case that indicates that it does not apply. Discard the results of all such pieces of pseudocode and their corresponding encoding diagrams.

   There is now one remaining piece of pseudocode and its corresponding encoding diagram left to consider. This pseudocode might also contain a special case, most commonly one indicating that it is CONSTRAINED UNPREDICTABLE. If so, abandon this execution model and treat the instruction according to the special case.
4. Check the *should be* bits of the encoding diagram against the corresponding bits of the bit pattern of the instruction. If any of them do not match, abandon this execution model and treat the instruction as CONSTRAINED UNPREDICTABLE.
5. Perform the rest of the operation pseudocode for the instruction description that contains the encoding diagram. That pseudocode starts with all variables set to the values they were left with by the encoding-specific pseudocode.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The `ConditionPassed()` call in the common pseudocode, if present, performs step 2, and the `EncodingSpecificOperations()` call performs steps 3 and 4.

H3.2.2 Limitations of the instruction pseudocode

The pseudocode descriptions of instruction functionality have a number of limitations. These are mainly due to the fact that, for clarity and brevity, the pseudocode is a sequential and mostly deterministic language.

These limitations include:

- Pseudocode does not describe the ordering requirements when an instruction generates multiple memory accesses. For a description of the ordering requirements on memory accesses, see External visibility requirement.
- Pseudocode does not describe the exact rules when an instruction that generates any of the following fails its condition code check:
  - UNDEFINED instruction.
  - Hyp trap.
  - Monitor trap.
  - Trap to AArch64 exception.

  In such cases, the `UNDEFINED` pseudocode statement or call to the applicable trap function lies inside the `if ConditionPassed() then ...` structure, either directly or in the `EncodingSpecificOperations()` function call, and so the pseudocode indicates that the instruction executes as a NOP. For the exact rules, see:
  - Conditional execution of undefined instructions.
  - EL2 configurable controls.
  - EL3 configurable controls.
  - Configurable instruction controls.
- Pseudocode does not describe the exact ordering requirements when a single floating-point instruction generates more than one floating-point exception and one or more of those floating-point exceptions is trapped. Combinations of floating-point exceptions describes the exact rules.

  Note

  There is no limitation in the case where all the floating-point exceptions are untrapped, because the pseudocode specifies the same behavior as the cross-referenced section.
- An exception can be taken during execution of the pseudocode for an instruction, either explicitly as a result of the execution of a pseudocode function such as `Abort()`, or implicitly, for example if an interrupt is taken during execution of an `LDM` instruction. If this happens, the pseudocode does not describe the extent to which the normal behavior of the instruction occurs.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H3.3 Data types

This section describes:

- General data type rules.
- Bitstrings.
- Integers.
- Reals.
- Booleans.
- Enumerations.
- Structures.
- Tuples.
- Arrays.

H3.3.1 General data type rules

Arm architecture pseudocode is a strongly typed language. Every literal and variable is of one of the following types:

- Bitstring.
- Integer.
- Boolean.
- Real.
- Enumeration.
- Tuple.
- Struct.
- Array.

The type of a literal is determined by its syntax. A variable can be assigned to without an explicit declaration. The variable implicitly has the type of the assigned value. For example, the following assignments implicitly declare the variables `x`, `y` and `z` to have types integer, bitstring of length 1, and Boolean, respectively.

```
x = 1;
y = '1';
z = TRUE;
```

Variables can also have their types declared explicitly by preceding the variable name with the name of the type. The following example declares explicitly that a variable named `count` is an integer.

```
integer count;
```

This is most often done in function definitions for the arguments and the result of the function.

The remaining subsections describe each data type in more detail.

H3.3.2 Bitstrings

This section describes the bitstring data type.

H3.3.2.1 Syntax

`bits(N)` The type name of a bitstring of length `N`.

`bit` A synonym of `bits(1)`.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H3.3.2.2 Description

A bitstring is a finite-length string of 0s and 1s. Each length of bitstring is a different type. The minimum permitted length of a bitstring is 0.

Bitstring constants literals are written as a single quotation mark, followed by the string of 0s and 1s, followed by another single quotation mark. For example, the two constants literals of type `bit` are `'0'` and `'1'`. Spaces can be included in bitstrings for clarity.

The bits in a bitstring are numbered from left to right N-1 to 0. This numbering is used when accessing the bitstring using bitslices. In conversions to and from integers, bit N-1 is the MSByte and bit 0 is the LSByte. This order matches the order in which bitstrings derived from encoding diagrams are printed.

Every bitstring value has a left-to-right order, with the bits being numbered in standard *little-endian* order. That is, the leftmost bit of a bitstring of length `N` is bit (N-1) and its right-most bit is bit 0. This order is used as the most-significant-to-least-significant bit order in conversions to and from integers. For bitstring constants and bitstrings that are derived from encoding diagrams, this order matches the way that they are printed.

Bitstrings are the only concrete data type in pseudocode, corresponding directly to the contents values that are manipulated in registers, memory locations, and instructions. All other data types are abstract.

H3.3.3 Integers

This section describes the data type for integer numbers.

H3.3.3.1 Syntax

`integer` The type name for the integer data type.

H3.3.3.2 Description

Pseudocode integers are unbounded in size and can be either positive or negative. That is, they are mathematical integers rather than what computer languages and architectures commonly call integers. Computer integers are represented in pseudocode as bitstrings of the appropriate length, and the pseudocode provides functions to interpret those bitstrings as integers.

Integer literals are normally written in decimal form, such as `0`, `15`, `-1234`. They can also be written in C-style hexadecimal form, such as `0x55` or `0x8000_0000`. Hexadecimal integer literals are treated as positive unless they have a preceding minus sign. For example, `0x8000_0000` is the integer $+2^{31}$. If -2^{31} needs to be written in hexadecimal, it must be written as `-0x8000_0000`.

H3.3.4 Reals

This section describes the data type for real numbers.

H3.3.4.1 Syntax

`real` The type name for the real data type.

H3.3.4.2 Description

Pseudocode reals are unbounded in size and precision. That is, they are mathematical real numbers, not computer floating-point numbers. Computer floating-point numbers are represented in pseudocode as bitstrings of the appropriate length, and the pseudocode provides functions to interpret those bitstrings as reals.

Real constant literals are written in decimal form with a decimal point. This means `0` is an integer constant literal, but `0.0` is a real constant literal.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H3.3.5 Booleans

This section describes the Boolean data type.

H3.3.5.1 Syntax

`boolean` The type name for the Boolean data type.

`TRUE or FALSE` The two values a Boolean variable can take.

H3.3.5.2 Description

A Boolean is a logical `TRUE` or `FALSE` value.

Note

This is not the same type as `bit`, which is a bitstring of length 1. A Boolean can only take on one of two values: `TRUE` or `FALSE`.

H3.3.6 Enumerations

This section describes the enumeration data type.

H3.3.6.1 Syntax and examples

`enumeration` Keyword to defined a new enumeration type.

```
enumeration Example {Example_One, Example_Two, Example_Three};
```

A definition of a new enumeration called `Example`, which can take on the values `Example_One`, `Example_Two`, `Example_Three`.

H3.3.6.2 Description

An enumeration is a defined set of named values.

An enumeration must contain at least one named value. A named value must not be shared between enumerations.

Enumerations must be defined explicitly, although a variable of an enumeration type can be declared implicitly by assigning one of the named values to it. By convention, each named value starts with the name of the enumeration followed by an underscore. The name of the enumeration is its *type name*, or *type*, and the named values are its possible *values*.

H3.3.7 Structures

This section describes the structure data type.

H3.3.7.1 Syntax and examples

`type` The keyword used to declare the structure data type.

```
type ShiftSpec is (bits(2) shift, integer amount)
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

An example definition for a new structure called `ShiftSpec` that contains an bitstring member called `shift` and a integer member named `amount`. Structure definitions must not be terminated with a semicolon.

```
ShiftSpec abc;
```

A declaration of a variable named `abc` of type `ShiftSpec`.

```
abc.shift
```

Syntax to refer to the individual members within the structure variable.

H3.3.7.2 Description

A structure is a compound data type composed of one or more data items. The data items can be of different data types. This can include compound data types. The data items of a structure are called its members and are named.

In the syntax section, the example defines a structure called `ShiftSpec` with two members. The first is a bitstring of length 2 named `shift` and the second is an integer named `amount`. After declaring a variable of that type named `abc`, the members of this structure are referred to as `abc.shift` and `abc.amount`.

Every definition of a structure creates a different type, even if the number and type of their members are identical. For example:

```
type ShiftSpec1 is (bits(2) shift, integer amount)
type ShiftSpec2 is (bits(2) shift, integer amount)
```

`ShiftSpec1` and `ShiftSpec2` are two different types despite having identical definitions. This means that the value in a variable of type `ShiftSpec1` cannot be assigned to variable of type `ShiftSpec2`.

H3.3.8 Tuples

This section describes the tuple data type.

H3.3.8.1 Examples

```
(bits(32) shifter_result, bit shifter_carry_out)
```

An example of the tuple syntax.

```
(shift_t, shift_n) = ('00', 0);
```

An example of assigning values to a tuple.

H3.3.8.2 Description

A tuple is an ordered set of data items, separated by commas and enclosed in parentheses. The items can be of different types and a tuple must contain at least one data item.

Tuples are often used as the return type for functions that return multiple results. For example, in the syntax section, the example tuple is the return type of the function `Shift_C()`, which performs a standard A32/T32 shift or rotation. Its return type is a tuple containing two data items, with the first of type `bits(32)` and the second of type `bit`.

Each tuple is a separate compound data type. The compound data type is represented as a comma-separated list of ordered data types between parentheses. This means that the example tuple at the start of this section is of type `(bits(32), bit)`. The general principle that types can be implied by an assignment extends to implying the type of the elements in the tuple. For example, in the syntax section, the example assignment implicitly declares:

- `shift_t` to be of type `bits(2)`.
- `shift_n` to be of type `integer`.
- `(shift_t, shift_n)` to be a tuple of type `(bits(2), integer)`.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H3.3.9 Arrays

This section describes the array data type.

H3.3.9.1 Syntax

`array` The type name for the array data type.

```
array data_type array_name[A..B];
```

Declaration of an array of type `data_type`, which might be compound data type. It is named `array_name` and is indexed with an integer range from `A` to `B`.

H3.3.9.2 Description

An array is an ordered set of fixed size containing items of a single data type. This can include compound data types. Pseudocode arrays are indexed by either enumerations or integer ranges. An integer range is represented by the lower inclusive end of the range, then `..`, then the upper inclusive end of the range.

For example:

The following example declares an array of 31 bitstrings of length 64, indexed from 0 to 30.

```
array bits(64) _R[0..30];
```

Arrays are always explicitly declared, and there is no notation for a constant literal array. Arrays always contain at least one element data item, because:

- Enumerations always contain at least one symbolic constant named value.
- Integer ranges always contain at least one integer.

An array declared with an enumeration type as the index must be accessed using enumeration values of that enumeration type. An array declared with an integer range type as the index must be accessed using integer values from that inclusive range. Accessing such an array with an integer value outside of the range is a coding error.

Arrays do not usually appear directly in pseudocode. The items that syntactically look like arrays in pseudocode are usually array-like functions such as `R[i]`, `MemU[address, size]` or `Elem[vector, i, size]`. These functions package up and abstract additional operations normally performed on accesses to the underlying arrays, such as register banking, memory protection, endian-dependent byte ordering, exclusive-access housekeeping and Advanced SIMD element processing. See Function and procedure calls.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H3.4 Operators

This section describes:

- Relational operators.
- Boolean operators.
- Bitstring operators.
- Arithmetic operators.
- The assignment operator.
- Precedence rules.
- Conditional expressions.
- Operator polymorphism.

H3.4.1 Relational operators

The following operations yield results of type `boolean`.

H3.4.1.1 Equality and non-equality

If two variables `x` and `y` are of the same type, their values can be tested for equality by using the expression `x == y` and for non-equality by using the expression `x != y`. In both cases, the result is of type `boolean`.

Both `x` and `y` must be of type `bits(N)`, `real`, `enumeration`, `boolean`, or `integer`. Named values from an `enumeration` can only be compared if they are both from the same `enumeration`. An exception is that a bitstring can be tested for equality with an integer to allow a `d==15` test.

A special form of comparison is defined with a bitstring literal that can contain bit values `'0'`, `'1'`, and `'x'`. Any bit with value `'x'` is ignored in determining the result of the comparison. For example, if `opcode` is a 4-bit bitstring, the expression `opcode == '1x0x'` matches the values `'1000'`, `'1100'`, `'1001'`, and `'1101'`. This is known as a bitmask.

Note

This special form is permitted in the implied equality comparisons in the `when` parts of `case ... of ...` structures.

H3.4.1.2 Comparisons

If `x` and `y` are integers or reals, then `x < y`, `x <= y`, `x > y`, and `x >= y` are less than, less than or equal, greater than, and greater than or equal comparisons between them, producing Boolean results.

H3.4.1.3 Set membership with `IN`

`<expression> IN {<set>}` produces `TRUE` if `<expression>` is a member of `<set>`. Otherwise, it is `FALSE`. `<set>` must be a list of expressions separated by commas.

H3.4.2 Boolean operators

If `x` is a Boolean expression, then `!x` is its logical inverse.

If `x` and `y` are Boolean expressions, then `x && y` is the result of ANDing them together. As in the C language, if `x` is `FALSE`, the result is determined to be `FALSE` without evaluating `y`.

Note

This is known as short circuit evaluation.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If `x` and `y` are `boolean`s, then `x || y` is the result of ORing them together. As in the C language, if `x` is `TRUE`, the result is determined to be `TRUE` without evaluating `y`.

Note

If `x` and `y` are `boolean`s or Boolean expressions, then the result of `x != y` is the same as the result of exclusive-ORing `x` and `y` together. The operator `EOR` only accepts bitstring arguments.

H3.4.3 Bitstring operators

The following operations can be applied only to bitstrings.

H3.4.3.1 Logical operations on bitstrings

If `x` is a bitstring, `NOT(x)` is the bitstring of the same length obtained by logically inverting every bit of `x`.

If `x` and `y` are bitstrings of the same length, `x AND y`, `x OR y`, and `x EOR y` are the bitstrings of that same length obtained by logically ANDing, logically ORing, and exclusive-ORing corresponding bits of `x` and `y` together.

H3.4.3.2 Bitstring concatenation and slicing

If `x` and `y` are bitstrings of lengths `N` and `M` respectively, then `x:y` is the bitstring of length `N+M` constructed by concatenating `x` and `y` in left-to-right order.

The bitstring slicing operator addresses specific bits in a bitstring. This can be used to create a new bitstring from extracted bits or to set the value of specific bits. Its syntax is `x<integer_list>`, where `x` is the integer or bitstring being sliced, and `<integer_list>` is a comma-separated list of integers enclosed in angle brackets. The length of the resulting bitstring is equal to the number of integers in `<integer_list>`. In `x<integer_list>`, each of the integers in `<integer_list>` must be:

- `>= 0`.
- `< Len(x)` if `x` is a bitstring.

The definition of `x<integer_list>` depends on whether `integer_list` contains more than one integer:

- If `integer_list` contains more than one integer, `x<i, j, k,..., n>` is defined to be the concatenation: `x<i> : x<j> : x<k> : ... : x<n>`.
- If `integer_list` consists of just one integer `i`, `x<i>` is defined to be:
  - If `x` is a bitstring, `'0'` if bit `i` of `x` is a zero and `'1'` if bit `i` of `x` is a one.
  - If `x` is an integer, and `y` is the unique integer in the range `0` to `2^(i+1)-1` that is congruent to `x` modulo `2^(i+1)`. Then `x<i>` is `'0'` if `y < 2^i` and `'1'` if `y >= 2^i`.
    Loosely, this definition treats an integer as equivalent to a sufficiently long two's complement representation of it as a bitstring.

The notation for a range expression is `i:j` with `i >= j` is shorthand for the integers in order from `i` down to `j`, with both end values included. For example, `instr<31:28>` represents `instr<31, 30, 29, 28>`.

`x<integer_list>` is assignable provided `x` is an assignable bitstring and no integer appears more than once in `<integer_list>`. In particular, `x<i>` is assignable if `x` is an assignable bitstring and `0 <= i < Len(x)`.

Encoding diagrams for registers frequently show named bits or multi-bit fields. For example, the encoding diagram for the APSR shows its bit<31> as `N`. In such cases, the syntax `APSR.N` is used as a more readable synonym for `APSR<31>` as named bits can be referred to with the same syntax as referring to members of a struct. A comma-separated list of named bits enclosed in angle brackets following the register name allows multiple bits to be addressed simultaneously. For example, `APSR.<N, C, Q>` is synonymous with `APSR <31, 29, 27>`.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H3.4.4 Arithmetic operators

Most pseudocode arithmetic is performed on integer or real values, with operands obtained by conversions from bitstrings and results converted back to bitstrings. As these data types are the unbounded mathematical types, no issues arise about overflow or similar errors.

H3.4.4.1 Unary plus and minus

If `x` is an integer or real, then `+x` is `x` unchanged, `-x` is `x` with its sign reversed. Both are of the same type as `x`.

H3.4.4.2 Addition and subtraction

If `x` and `y` are integers or reals, `x+y` and `x-y` are their sum and difference. Both are of type `integer` if `x` and `y` are both of type `integer`, and `real` otherwise.

There are two cases where the types of `x` and `y` can be different. A bitstring and an integer can be added together to allow the operation `PC + 4`. An integer can be subtracted from a bitstring to allow the operation `PC - 2`.

If `x` and `y` are bitstrings of the same length `N`, so that `N = Len(x) = Len(y)`, then `x+y` and `x-y` are the least significant *N* bits of the results of converting `x` and `y` to integers and adding or subtracting them. Signed and unsigned conversions produce the same result:

```
x+y = (SInt(x) + SInt(y))<N-1:0>
    = (UInt(x) + UInt(y))<N-1:0>
x-y = (SInt(x) - SInt(y))<N-1:0>
    = (UInt(x) - UInt(y))<N-1:0>
```

If `x` is a bitstring of length `N` and `y` is an integer, `x+y` and `x-y` are the bitstrings of length `N` defined by `x+y = x + y<N-1:0>` and `x-y = x - y<N-1:0>`. Similarly, if `x` is an integer and `y` is a bitstring of length `M`, `x+y` and `x-y` are the bitstrings of length `M` defined by `x+y = x<M-1:0> + y` and `x-y = x<M-1:0> - y`.

H3.4.4.3 Multiplication

If `x` and `y` are integers or reals, then `x * y` is the product of `x` and `y`. It is of type `integer` if `x` and `y` are both of type `integer`, and `real` otherwise.

H3.4.4.4 Division and modulo

If `x` and `y` are reals, then `x/y` is the result of dividing `x` by `y`, and is always of type `real`.

If `x` and `y` are integers, then `x DIV y` and `x MOD y` are defined by:

```
x DIV y = RoundDown(x/y)
x MOD y = x - y * (x DIV y)
```

It is a pseudocode error to use any of `x/y`, `x MOD y`, or `x DIV y` in any context where `y` can be zero.

H3.4.4.5 Scaling

If `x` and `n` are of type `integer`, then:

- `x << n = RoundDown(x * 2^n)`.
- `x >> n = RoundDown(x * 2^(-n))`.

H3.4.4.6 Raising to a power

If `x` is an integer or a real and `n` is an integer, then `x^n` is the result of raising `x` to the power of `n`, and:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If `x` is of type `integer`, then `x^n` is of type `integer`.
- If `x` is of type `real`, then `x^n` is of type `real`.

H3.4.5 The assignment operator

The assignment operator is the = character, which assigns the value of the right-hand side to the left-hand side. An assignment statement takes the form:

```
<assignable_expression> = <expression>;
```

This following subsection defines valid expression syntax.

H3.4.5.1 General expression syntax

An expression is one of the following:

- A literal.
- A variable, optionally preceded by a data type name to declare its type.
- The word `UNKNOWN` preceded by a data type name to declare its type.
- The result of applying a language-defined operator to other expressions.
- The result of applying a function to other expressions.

Variable names normally consist of alphanumeric and underscore characters, starting with an alphabetic or underscore character.

Each register defined in an Arm architecture specification defines a correspondingly named pseudocode bitstring variable, and that variable has the stated behavior of the register. For example, if a bit of a register is defined as RAZ/WI, then the corresponding bit of its variable reads as '0' and ignore writes.

An expression like `bits(32) UNKNOWN` indicates that the result of the expression is a value of the given type, but the architecture does not specify what value it is and software must not rely on such values. The value produced must not:

- Return information that cannot be accessed at the current or a lower level of privilege using instructions that are not UNPREDICTABLE or `CONSTRAINED UNPREDICTABLE` and do not return UNKNOWN values,
- Be promoted as providing any useful information to software.

Note

UNKNOWN values are similar to the definition of UNPREDICTABLE, but do not indicate that the entire architectural state becomes unspecified.

Only the following expressions are assignable. This means that these are the only expressions that can be placed on the left-hand side of an assignment.

- Variables.
- The results of applying some operators to other expressions.

  The description of each language-defined operator that can generate an assignable expression specifies the circumstances under which it does so. For example, those circumstances might require that one or more of the expressions the operator operates on is an assignable expression.
- The results of applying array-like functions to other expressions. The description of an array-like function specifies the circumstances under which it can generate an assignable expression.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

If the right-hand side in an assignment is a function returning a tuple, an item in the assignment destination can be written as – to indicate that the corresponding item of the assigned tuple value is discarded. For example:

```
(shifted, -) = LSL_C(operand, amount);
```

The expression on the right-hand side itself can be a tuple. For example:

```
(x, y) = (function_1(), function_2());
```

Every expression has a data type.

- For a literal, this data type is determined by the syntax of the literal.
- For a variable, there are the following possible sources for the data type
  - An optional preceding data type name.
  - A data type the variable was given earlier in the pseudocode by recursive application of this rule.
  - A data type the variable is being given by assignment, either by direct assignment to the variable, or by assignment to a list of which the variable is a member.

  It is a pseudocode error if none of these data type sources exists for a variable, or if more than one of them exists and they do not agree about the type.
- For a language-defined operator, the definition of the operator determines the data type.
- For a function, the definition of the function determines the data type.

H3.4.6 Precedence rules

The precedence rules for expressions are:

1. Literals, variables and function invocations are evaluated with higher priority than any operators using their results, but see Boolean operators.
2. Operators on integers follow the normal operator precedence rules of *exponentiation before multiply/divide before add/subtract*, with sequences of multiply/divides or add/subtracts evaluated left-to-right.
3. Other expressions must be parenthesized to indicate operator precedence if ambiguity is possible, but need not be if all permitted precedence orders under the type rules necessarily lead to the same result. For example, if `i`, `j` and `k` are integer variables, `i > 0 && j > 0 && k > 0` is acceptable, but `i > 0 && j > 0 || k > 0` is not.

H3.4.7 Conditional expressions

If `x` and `y` are two values of the same type and `t` is a value of type `boolean`, then `if t then x else y` is an expression of the same type as `x` and `y` that produces `x` if `t` is `TRUE` and `y` if `t` is `FALSE`.

H3.4.8 Operator polymorphism

Operators in pseudocode can be polymorphic, with different functionality when applied to different data types. Each resulting form of an operator has a different prototype definition. For example, the operator + has forms that act on various combinations of integers, reals and bitstrings.

Table H3-1 summarizes the operand types valid for each unary operator and the result type. Table H3-2 summarizes the operand types valid for each binary operator and the result type.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table H3-1 Result and operand types permitted for unary operators

Operator	Operand Type	Result Type
-	integer	integer
	real	real
NOT	bits(N)	bits(N)
!	boolean	boolean

Table H3-2 Result and operand types permitted for binary operators

Operator	First operand type	Second operand type	Result type
==	bits(N)	integer	boolean
		bits(N)	
	integer	integer	
	real	real	
	enumeration	enumeration	
	boolean	boolean	
!=	bits(N)	bits(N)	boolean
	integer	integer	
	real	real	
<, > <=, >=	integer	integer	boolean
	real	real	
+, -	integer	integer	integer
	real	real	real
	bits(N)	bits(N)	bits(N)
		integer	
<<, >>	integer	integer	integer
*	integer	integer	integer
	real	real	real
	bits(N)	bits(N)	bits(N)
/	real	real	real
DIV	integer	integer	integer
MOD	integer	integer	integer
	bits(N)	integer	
&&, ||	boolean	boolean	boolean
AND, OR, EOR	bits(N)	bits(N)	bits(N)
^	integer	integer	integer
	real	integer	real

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H3.5 Statements and control structures

This section describes the statements and program structures available in the pseudocode:

- Statements and Indentation.
- Function and procedure calls.
- Conditional control structures.
- Loop control structures.
- Special statements.
- Comments.

H3.5.1 Statements and Indentation

A simple statement is either an assignment, a function call, or a procedure call. Each statement must be terminated with a semicolon.

Indentation normally indicates the structure in compound statements. The statements contained in structures such as `if ... then ... else ...` or procedure and function definitions are indented more deeply than the statement structure itself. The end of a compound statement structure and their end is indicated by returning to the original indentation level or less.

Indentation is normally done by four spaces for each level. Standard indentation uses four spaces for each level of indent.

H3.5.2 Function and procedure calls

This section describes how functions and procedures are defined and called in the pseudocode.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H3.5.2.1 Procedure and function definitions

A procedure definition has the form:

```
<procedure name>(<argument prototypes>)
    <statement 1>;
    <statement 2>;
    ...
    <statement n>;
```

where `<argument prototypes>` consists of zero or more argument definitions, separated by commas. Each argument definition consists of a type name followed by the name of the argument.

Note

This first definition line is not terminated by a semicolon. This distinguishes it from a procedure call.

A function definition is similar, but also declares the return type of the function:

```
<return type> <function name>(<argument prototypes>)
    <statement 1>;
    <statement 2>;
    ...
    <statement n>;
```

Note

A function or procedure name can include a ".". This is a convention used for functions that have similar but different behaviors in AArch32 and AArch64 states.

Array-like functions are similar, but are written with square brackets and have two forms. These two forms exist because reading from and writing to an array element require different functions. They are frequently used in memory operations. An array-like function definition with a return type is equivalent to reading from an array. For example:

```
<return type> <function name>[<argument prototypes>]
    <statement 1>;
    <statement 2>;
    ...
    <statement n>;
```

Its related function definition with no return type is equivalent to writing to an array. For example:

```
<function name>[<argument prototypes>] = <value prototype>
    <statement 1>;
    <statement 2>;
    ...
    <statement n>;
```

The value prototype determines what data type can be written to the array. The two related functions must share the same name, but the value prototype and return type can be different.

H3.5.2.2 Procedure calls

A procedure call has the form:

```
<procedure_name>(<arguments>);
```

H3.5.2.3 Return statements

A procedure return has the form:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
return;
```

A function return has the form:

```
return <expression>;
```

where `<expression>` is of the type declared in the function prototype line.

H3.5.3 Conditional control structures

This section describes how conditional control structures are used in the pseudocode.

H3.5.3.1 if . . . then . . . else . . .

In addition to being a ternary operator, a multi-line `if ... then ... else ...` structure can act as a control structure and has the form:

```
if <boolean_expression> then
    <statement 1>;
    <statement 2>;
    ...
    <statement n>;

elsif <boolean_expression> then
    <statement a>;
    <statement b>;
    ...
    <statement z>;
else
    <statement A>;
    <statement B>;
    ...
    <statement Z>;
```

The block of lines consisting of `elsif` and its indented statements is optional, and multiple `elsif` blocks can be used.

The block of lines consisting of `else` and its indented statements is optional.

Abbreviated one-line forms can be used when the `then` part, and in the `else` part if it is present, contain only simple statements such as:

```
if <boolean_expression> then <statement 1>;
if <boolean_expression> then <statement 1>; else <statement A>;
if <boolean_expression> then <statement 1>; <statement 2>; else <statement A>;
```

Note

In these forms, `<statement 1>`, `<statement 2>`, and `<statement A>` must be terminated by semicolons. This, and the fact that the `else` part is optional, distinguish its use as a control structure from its use as a ternary operator.

H3.5.3.2 case . . . of . . .

A `case ... of ...` structure has the form:

```
case <expression> of
    when <literal values1>
        <statement 1>;
        <statement 2>;
        ...
        <statement n>;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    when <literal values2>
        <statement 1>;
        <statement 2>;
        ...
        <statement n>;

    ... more "when" groups if required ...

    otherwise
        <statement A>;
        <statement B>;
        ...
        <statement Z>;
```

In this structure, `<literal values1>` and `<literal values2>` consist of literal values of the same type as `<expression>`, separated by commas. There can be additional `when` groups in the structure. Abbreviated one line forms of `when` and `otherwise` parts can be used when they contain only simple statements.

If `<expression>` has a bitstring type, the literal values can also include bitstring literals containing `'x'` bits, known as bitmasks. For details, see Equality and non-equality.

H3.5.4 Loop control structures

This section describes the three loop control structures used in the pseudocode.

H3.5.4.1 repeat . . . until . . .

A `repeat ... until ...` structure has the form:

```
repeat
    <statement 1>;
    <statement 2>;
    ...
    <statement n>;
until <boolean_expression>;
```

It executes the statement block at least once, and the loop repeats until `<boolean expression>` evaluates to `TRUE`. Variables explicitly declared inside the loop body have scope local to that loop and might not be accessed outside the loop body.

H3.5.4.2 while . . . do

A `while ... do` structure has the form:

```
while <boolean_expression> do
    <statement 1>;
    <statement 2>;
    ...
    <statement n>;
```

It begins executing the statement block only if the Boolean expression is true. The loop then runs until the expression is false.

H3.5.4.3 for . . .

A `for ...` structure has the form:

```
for <assignable_expression> = <integer_expr1> to <integer_expr2>
    <statement 1>;
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    <statement 2>;
    ...
    <statement n>;
```

The <assignable_expression> is initialized to <integer_expr1> and compared to <integer_expr2>. If <assignable_expression> is less than or equal to <integer_expr2>, the loop body is executed and the <assignable_expression> incremented by one. The comparison, execution of the loop body, and increment is repeated until <assignable_expression> is greater than <integer_expr2>.

There is an alternate form:

```
for <assignable_expression> = <integer_expr1> downto <integer_expr2>
```

where <assignable_expression> is decremented after the comparison, execution of the loop body, and decrement repeats until <assignable_expression> is less than <integer_expr2>.

H3.5.5 Special statements

This section describes statements with particular architecturally defined behaviors.

H3.5.5.1 UNDEFINED

This subsection describes the statement:

```
UNDEFINED;
```

This statement indicates a special case that replaces the behavior defined by the current pseudocode, apart from behavior required to determine that the special case applies. The replacement behavior is that the Undefined Instruction exception is taken.

H3.5.5.2 UNPREDICTABLE

This subsection describes the statement:

```
UNPREDICTABLE;
```

This statement indicates a special case that replaces the behavior defined by the current pseudocode, apart from behavior required to determine that the special case applies. The replacement behavior is UNPREDICTABLE.

H3.5.5.3 SEE. . .

This subsection describes the statement:

```
SEE <reference>;
```

This statement indicates a special case that replaces the behavior defined by the current pseudocode, apart from behavior required to determine that the special case applies. The replacement behavior is that nothing occurs as a result of the current pseudocode because some other piece of pseudocode defines the required behavior. The <reference> indicates where that other pseudocode can be found.

It usually refers to another instruction, but can also refer to another encoding or note of the same instruction.

H3.5.5.4 IMPLEMENTATION_DEFINED

This subsection describes the statement:

```
IMPLEMENTATION_DEFINED {"<text>"};
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This statement indicates a special case that replaces the behavior defined by the current pseudocode, apart from behavior required to determine that the special case applies. The replacement behavior is IMPLEMENTATION DEFINED. An optional `<text>` field can give more information.

H3.5.6 Comments

The pseudocode supports two styles of comments:

- `//` starts a comment that is terminated by the end of the line.
- `/*` starts a comment that is terminated by `*/`.

`/**/` statements might not be nested, and the first `*/` ends the comment.

Note

Comment lines do not require a terminating semicolon.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H3.6 Built-in functions

This section describes:

- Bitstring manipulation functions.
- Arithmetic functions.

H3.6.1 Bitstring manipulation functions

The following bitstring manipulation functions are defined:

H3.6.1.1 Bitstring length and most significant bit

If `x` is a bitstring:

- The bitstring length function `Len(x)` returns the length of `x` as an integer.

H3.6.1.2 Bitstring concatenation and replication

If `x` is a bitstring and `n` is an integer with `n >= 0`:

- `Replicate(x, n)` is the bitstring of length `n*Len(x)` consisting of `n` copies of `x` concatenated together.
- `Zeros(n) = Replicate('0', n)`.
- `Ones(n) = Replicate('1', n)`.

H3.6.1.3 Bitstring count

If `x` is a bitstring, `BitCount(x)` is an integer result equal to the number of bits of `x` that are ones.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H3.6.1.4 Testing a bitstring for being all zero or all ones

If `x` is a bitstring:

- `IsZero(x)` produces `TRUE` if all of the bits of `x` are zeros and `FALSE` if any of them are ones
- `IsZeroBit(x)` produces `'1'` if all of the bits of `x` are zeros and `'0'` if any of them are ones.

`IsOnes(x)` and `IsOnesBit(x)` work in the corresponding ways. This means:

```
IsZero(x)    = (BitCount(x) == 0)
IsOnes(x)    = (BitCount(x) == Len(x))
IsZeroBit(x) = if IsZero(x) then '1' else '0'
IsOnesBit(x) = if IsOnes(x) then '1' else '0'
```

H3.6.1.5 Lowest and highest set bits of a bitstring

If `x` is a bitstring, and `N = Len(x)`:

- `LowestSetBit(x)` is the minimum bit number of any of the bits of `x` that are ones. If all of its bits are zeros, `LowestSetBit(x) = N`.
- `HighestSetBit(x)` is the maximum bit number of any of the bits of `x` that are ones. If all of its bits are zeros, `HighestSetBit(x) = -1`.
- `CountLeadingZeroBits(x)` is the number of zero bits at the left end of x, in the range 0 to *N*. This means: `CountLeadingZeroBits(x) = N - 1 - HighestSetBit(x)`.
- `CountLeadingSignBits(x)` is the number of copies of the sign bit of x at the left end of x, excluding the sign bit itself, and is in the range 0 to *N*-1. This means: `CountLeadingSignBits(x) = CountLeadingZeroBits(x<N-1:1> EOR x<N-2:0>)`.

H3.6.1.6 Zero-extension and sign-extension of bitstrings

If `x` is a bitstring and `i` is an integer, then `ZeroExtend(x, i)` is `x` extended to a length of `i` bits, by adding sufficient zero bits to its left. That is, if `i == Len(x)`, then `ZeroExtend(x, i) = x`, and if `i > Len(x)`, then:

```
ZeroExtend(x, i) = Replicate('0', i-Len(x)) : x
```

If `x` is a bitstring and `i` is an integer, then `SignExtend(x, i)` is `x` extended to a length of `i` bits, by adding sufficient copies of its leftmost bit to its left. That is, if `i == Len(x)`, then `SignExtend(x, i) = x`, and if `i > Len(x)`, then:

```
SignExtend(x, i) = Replicate(TopBit(x), i-Len(x)) : x
```

It is a pseudocode error to use either `ZeroExtend(x, i)` or `SignExtend(x, i)` in a context where it is possible that `i < Len(x)`.

H3.6.1.7 Converting bitstrings to integers

If `x` is a bitstring, `SInt()` is the integer whose two's complement representation is `x`.

`UInt()` is the integer whose unsigned representation is `x`.

`Int (x, unsigned)` returns either `SInt(x)` or `UInt(x)` depending on the value of its second argument.

H3.6.2 Arithmetic functions

This section defines built-in arithmetic functions.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H3.6.2.1 Absolute value

If `x` is either of type real or integer, `Abs(x)` returns the absolute value of `x`. The result is the same type as `x`.

H3.6.2.2 Rounding and aligning

If `x` is a real:

- `RoundDown(x)` produces the largest integer `n` such that `n <= x`.
- `RoundUp(x)` produces the smallest integer `n` such that `n >= x`.
- `RoundTowardsZero(x)` produces:
  - `RoundDown(x)` if `x > 0.0`.
  - `0` if `x == 0.0`.
  - `RoundUp(x)` if `x < 0.0`.

If `x` and `y` are both of type `integer`, `Align(x, y) = y * (x DIV y)`, and is of type `integer`.

If `x` is of type `bitstring` and `y` is of type `integer`, `Align(x, y) = (Align(UInt(x), y))<Len(x)-1:0>`, and is a `bitstring` of the same length as `x`.

It is a pseudocode error to use either form of `Align(x, y)` in any context where `y` can be 0. In practice, `Align(x, y)` is only used with `y` a constant power of two, and the bitstring form used with `y = 2^n` has the effect of producing its argument with its n low-order bits forced to zero.

H3.6.2.3 Maximum and minimum

If `x` and `y` are integers or reals, then `Max(x, y)` and `Min(x, y)` are their maximum and minimum respectively. x and y must both be of type integer or of type real. The function returns a value of the same type as its operands.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H3.7 Miscellaneous helper procedures and functions

This section lists the prototypes of miscellaneous *helper* procedures and functions used by the pseudocode, together with a brief description of the effect of the procedure or function. The pseudocode does not define the operation of these helper procedures and functions.

Note

A-profile Architecture Pseudocode also has an entry for each of these functions, but currently these entries do not say anything about the effect of the function. When this information is added in A-profile Architecture Pseudocode, this section will be removed from the manual.

H3.7.1 EndOfInstruction()

This procedure terminates processing of the current instruction.

```
EndOfInstruction();
```

H3.7.2 Hint_Debug()

This procedure supplies a hint to the debug system.

```
Hint_Debug(bits(4) option);
```

H3.7.3 Hint_PreloadData()

This procedure performs a *preload data* hint.

```
Hint_PreloadData(bits(32) address);
```

H3.7.4 Hint_PreloadDataForWrite()

This procedure performs a *preload data* hint with a probability that the use will be for a write.

```
Hint_PreloadDataForWrite(bits(32) address);
```

H3.7.5 Hint_PreloadInstr()

This procedure performs a *preload instructions* hint.

```
Hint_PreloadInstr(bits(32) address);
```

H3.7.6 Hint_Yield()

This procedure performs a *Yield* hint.

```
Hint_Yield();
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H3.7.7 IsExternalAbort()

This function returns `TRUE` if the abort currently being processed is an External abort and FALSE otherwise. It is used only in exception entry pseudocode.

```
boolean IsExternalAbort(Fault type)
    assert type != Fault_None;

boolean IsExternalAbort(FaultRecord fault);
```

H3.7.8 IsAsyncAbort()

This function returns `TRUE` if the abort currently being processed is an asynchronous abort, and `FALSE` otherwise. It is used only in exception entry pseudocode.

```
boolean IsAsyncAbort(Fault type)
    assert type != Fault_None;

boolean IsAsyncAbort(FaultRecord fault);
```

H3.7.9 LSInstructionSyndrome()

This function returns the extended syndrome information for a fault reported in the HSR.

```
bits(11) LSInstructionSyndrome();
```

H3.7.10 ProcessorID()

This function returns an integer that uniquely identifies the executing PE in the system.

```
integer ProcessorID();
```

H3.7.11 RemapRegsHaveResetValues()

This function returns `TRUE` if the remap registers PRRR and NMRR have their IMPLEMENTATION DEFINED reset values, and `FALSE` otherwise.

```
boolean RemapRegsHaveResetValues();
```

H3.7.12 ResetControlRegisters()

This function resets the System registers and memory-mapped control registers that have architecturally defined reset values to those values. For more information about the affected registers, see:

- Reset behavior.

```
AArch64.ResetControlRegisters(boolean ResetIsCold)
```

H3.7.13 ThisInstr()

This function returns the bitstring encoding of the currently executing instruction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
bits(32) ThisInstr();
```

Note

Currently, this function is used only on 32-bit instruction encodings.

H3.7.14 ThisInstrLength()

This function returns the length, in bits, of the current instruction. This means it returns 32 or 16:

```
integer ThisInstrLength();
```


Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H3.8 Arm pseudocode definition index

This section contains the following tables:

- Table H3-3 which contains the pseudocode data types.
- Table H3-4 which contains the pseudocode operators.
- Table H3-5 which contains the pseudocode keywords and control structures.
- Table H3-6 which contains the statements with special behaviors.

Table H3-3 Index of pseudocode data types

Keyword	Meaning
`array`	Type name for the array type
`bit`	Keyword equivalent to `bits(1)`
`bits(N)`	Type name for the bitstring of length `N` data type
`boolean`	Type name for the Boolean data type
`enumeration`	Keyword to define a new enumeration type
`integer`	Type name for the integer data type
`real`	Type name for the real data type
`type`	Keyword to define a new structure

Table H3-4 Index of pseudocode operators

Operator	Meaning
`-`	Unary minus on integers or reals
	Subtraction of integers, reals, and bitstrings
	Used in the left-hand side of an assignment or a tuple to discard the result
`+`	Unary plus on integers or reals
	Addition of integers, reals, and bitstrings
`.`	Extract named member from a list
	Extract named bit or field from a register
`:`	Bitstring concatenation
	Integer range in bitstring extraction operator
`!`	Boolean NOT
`!=`	Comparison for inequality
`(...)`	Around arguments of procedure or function
`[...]`	Around array index
	Around arguments of array-like function
`*`	Multiplication of integers, reals, and bitstrings
`/`	Division of reals
`&&`	Boolean AND

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operator	Meaning
`<`	*Less than* comparison of integers and reals
`<...>`	Slicing of specified bits of bitstring or integer
`<<`	Multiply integer by power of 2
`<=`	*Less than or equal* comparison of integers and reals
`=`	Assignment operator
`==`	Comparison for equality
`>`	*Greater than* comparison of integers and reals
`>=`	*Greater than or equal* comparison of integers and reals
`>>`	Divide integer by power of 2
`||`	Boolean OR
`^`	Exponential operator
`AND`	Bitwise AND of bitstrings
`DIV`	Quotient from integer division
`EOR`	Bitwise EOR of bitstrings
`IN`	Tests membership of a certain expression in a set of values
`MOD`	Remainder from integer division
`NOT`	Bitwise inversion of bitstrings
`OR`	Bitwise OR of bitstrings
`case ... of ...`	Control structure for the
`if ... then ... else ...`	Condition expression selecting between two values

Table H3-5 Index of pseudocode keywords and control structures

Operator	Meaning
`/*...*/`	Comment delimiters
`//`	Introduces comment terminated by end of line
`FALSE`	One of two values a Boolean can take (other than `TRUE`)
`for ... = ...to ...`	Loop control structure, counting up from the initial value to the upper limit
`for ... = ... downto ...`	Loop control structure, counting down from the initial value to the lower limit
`if ... then ... else ...`	Conditional control structure
`otherwise`	Introduces default case in `case ... of ...` control structure
`repeat ... until ...`	Loop control structure that runs at least once until the termination condition is satisfied
`return`	Procedure or function return
`TRUE`	One of two values a Boolean can take (other than `FALSE`)
`when`	Introduces specific case in `case ... of ...` control structure
`while ... do ...`	Loop control structure that runs until the termination condition is satisfied

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table H3-6 Index of special statements

Keyword	Meaning
`IMPLEMENTATION_DEFINED`	Describes IMPLEMENTATION DEFINED behavior
`SEE`	Points to other pseudocode to use instead
`UNDEFINED`	Cause Undefined Instruction exception
`UNKNOWN`	Unspecified value
`UNPREDICTABLE`	Unspecified behavior

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Appendix H4
Registers Index

This appendix provides indexes to the register descriptions in this manual. It contains the following sections:

- Introduction and register disambiguation.
- Alphabetical index of AArch64 registers and System instructions.
- Functional index of AArch64 registers and System instructions.
- Alphabetical index of memory-mapped registers.
- Functional index of memory-mapped registers.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H4.1 Introduction and register disambiguation

In some sections of this manual, registers are referred to by a *general name*, where the description applies to more than one context. In an AArch64 only implementation the description applies to multiple Exception levels, and therefore at a particular Exception level the register names need to take the appropriate Exception level suffix, _EL0, _EL1, or _EL2.

The following section disambiguates the general register names:

- Register name disambiguation by Exception level.

H4.1.1 Register name disambiguation by Exception level

Table H4-1 disambiguates the general names of the AArch64 System registers by Exception level.

Table H4-1 Disambiguation of AArch64 System registers by Exception level

General form	EL0	EL1	EL2
AFSR0_ELx	-	AFSR0_EL1	AFSR0_EL2
AFSR1_ELx	-	AFSR1_EL1	AFSR1_EL2
AMAIR_ELx	-	AMAIR_EL1	AMAIR_EL2
CONTEXTIDR_ELx	-	CONTEXTIDR_EL1	CONTEXTIDR_EL2
CPTR_ELx	-	-	CPTR_EL2
ELR_ELx	-	ELR_EL1	ELR_EL2
ESR_ELx	-	ESR_EL1	ESR_EL2
FAR_ELx	-	FAR_EL1	FAR_EL2
MAIR_ELx	-	MAIR_EL1	MAIR_EL2
MPUIR_ELx	-	MPUIR_EL1	MPUIR_EL2
PRBAR_ELx	-	PRBAR_EL1	PRBAR_EL2
PRBARn_ELx	-	PRBARn_EL1	PRBARn_EL2
PRENR_ELx	-	PRENR_EL1	PRENR_EL2
PRLAR_ELx	-	PRLAR_EL1	PRLAR_EL2
PRLARn_ELx	-	PRLARn_EL1	PRLARn_EL2
PRSELR_ELx	-	PRSELR_EL1	PRSELR_EL2
RMR_ELx	-	-	RMR_EL2
RVBAR_ELx	-	-	RVBAR_EL2
SCXTNUM_ELx	SCXTNUM_EL0	SCXTNUM_EL1	SCXTNUM_EL2
SP_ELx	SP_EL0	SP_EL1	SP_EL2
SPSR_ELx	-	SPSR_EL1	SPSR_EL2
TCR_ELx	-	TCR_EL1	TCR_EL2
TRFCR_ELx	-	TRFCR_EL1	TRFCR_EL2
TTBR_ELx	-	TTBR0_EL1, TTBR1_EL1	-
TTBRn_ELx	-	TTBR0_EL1, TTBR1_EL1	-

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

General form	EL0	EL1	EL2
TTBR1_ELx	-	TTBR1_EL1	-
VBAR_ELx	-	VBAR_EL1	VBAR_EL2

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H4.2 Alphabetical index of AArch64 registers and System instructions

This section is an index of AArch64 registers and System instructions in alphabetical order.

Table H4-2 Alphabetical index of AArch64 Registers

Register	Description, see
ACTLR_EL1	Auxiliary Control Register (EL1)
ACTLR_EL2	Auxiliary Control Register (EL2)
AFSR0_EL1	Auxiliary Fault Status Register 0 (EL1)
AFSR0_EL2	Auxiliary Fault Status Register 0 (EL2)
AFSR1_EL1	Auxiliary Fault Status Register 1 (EL1)
AFSR1_EL2	Auxiliary Fault Status Register 1 (EL2)
AIDR_EL1	Auxiliary ID Register
AMAIR_EL1	Auxiliary Memory Attribute Indirection Register (EL1)
AMAIR_EL2	Auxiliary Memory Attribute Indirection Register (EL2)
APDAKeyHi_EL1	Pointer Authentication Key A for Data (bits[127:64])
APDAKeyLo_EL1	Pointer Authentication Key A for Data (bits[63:0])
APDBKeyHi_EL1	Pointer Authentication Key B for Data (bits[127:64])
APDBKeyLo_EL1	Pointer Authentication Key B for Data (bits[63:0])
APGAKeyHi_EL1	Pointer Authentication Key A for Code (bits[127:64])
APGAKeyLo_EL1	Pointer Authentication Key A for Code (bits[63:0])
APIAKeyHi_EL1	Pointer Authentication Key A for Instruction (bits[127:64])
APIAKeyLo_EL1	Pointer Authentication Key A for Instruction (bits[63:0])
APIBKeyHi_EL1	Pointer Authentication Key B for Instruction (bits[127:64])
APIBKeyLo_EL1	Pointer Authentication Key B for Instruction (bits[63:0])
AT S12E0R	Address Translate Stages 1 and 2 EL0 Read
AT S12E0W	Address Translate Stages 1 and 2 EL0 Write
AT S12E1R	Address Translate Stages 1 and 2 EL1 Read
AT S12E1W	Address Translate Stages 1 and 2 EL1 Write
AT S1E0R	Address Translate Stage 1 EL0 Read
AT S1E0W	Address Translate Stage 1 EL0 Write
AT S1E1R	Address Translate Stage 1 EL1 Read
AT S1E1RP	Address Translate Stage 1 EL1 Read PAN
AT S1E1W	Address Translate Stage 1 EL1 Write
AT S1E1WP	Address Translate Stage 1 EL1 Write PAN
AT S1E2R	Address Translate Stage 1 EL2 Read
AT S1E2W	Address Translate Stage 1 EL2 Write
AT S1E3R	Address Translate Stage 1 EL3 Read
AT S1E3W	Address Translate Stage 1 EL3 Write
CCSIDR2_EL1	Current Cache Size ID Register 2
CCSIDR_EL1	Current Cache Size ID Register
CFP RCTX	Control Flow Prediction Restriction by Context

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
CLIDR_EL1	Cache Level ID Register
CNTFRQ_EL0	Counter-timer Frequency Register
CNTHCTL_EL2	Counter-timer Hypervisor Control Register
CNTHPS_CTL_EL2	Counter-timer Secure Physical Timer Control Register (EL2)
CNTHPS_CVAL_EL2	Counter-timer Secure Physical Timer CompareValue Register (EL2)
CNTHPS_TVAL_EL2	Counter-timer Secure Physical Timer TimerValue Register (EL2)
CNTKCTL_EL1	Counter-timer Kernel Control Register
CNTP_CTL_EL0	Counter-timer Physical Timer Control Register
CNTP_CVAL_EL0	Counter-timer Physical Timer CompareValue Register
CNTP_TVAL_EL0	Counter-timer Physical Timer TimerValue Register
CNTPCT_EL0	Counter-timer Physical Count Register
CNTV_CTL_EL0	Counter-timer Virtual Timer Control Register
CNTV_CVAL_EL0	Counter-timer Virtual Timer CompareValue Register
CNTV_TVAL_EL0	Counter-timer Virtual Timer TimerValue Register
CNTVCT_EL0	Counter-timer Virtual Count Register
CNTVOFF_EL2	Counter-timer Virtual Offset Register
CONTEXTIDR_EL1	Context ID Register (EL1)
CONTEXTIDR_EL2	Context ID Register (EL2)
CPACR_EL1	Architectural Feature Access Control Register
CPP RCTX	Cache Prefetch Prediction Restriction by Context
CPTR_EL2	Architectural Feature Trap Register (EL2)
CSSELR_EL1	Cache Size Selection Register
CTR_EL0	Cache Type Register
CurrentEL	Current Exception Level
DAIF	Interrupt Mask Bits
DBGAUTHSTATUS_EL1	Debug Authentication Status Register
DBGCLAIMCLR_EL1	Debug CLAIM Tag Clear Register
DBGCLAIMSET_EL1	Debug CLAIM Tag Set Register
DBGDTR_EL0	Debug Data Transfer Register, half-duplex
DBGDTRRX_EL0	Debug Data Transfer Register, Receive
DBGDTRTX_EL0	Debug Data Transfer Register, Transmit
DBGPRCR_EL1	Debug Power Control Register
DC CISW	Data or unified Cache line Clean and Invalidate by Set-Way
DC CIVAC	Data or unified Cache line Clean and Invalidate by VA to PoC
DC CSW	Data or unified Cache line Clean by Set-Way
DC CVAC	Data or unified Cache line Clean by VA to PoC
DC CVADP	Data or unified Cache line Clean by VA to PoDP
DC CVAP	Data or unified Cache line Clean by VA to PoP
DC CVAU	Data or unified Cache line Clean by VA to PoU
DC ISW	Data or unified Cache line Invalidate by Set-Way

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
DC IVAC	Data or unified Cache line Invalidate by VA to PoC
DC ZVA	Data Cache Zero by VA
DCZID_EL0	Data Cache Zero ID Register
DISR_EL1	Deferred Interrupt Status Register
DIT	Data Independent Timing
DLR_EL0	Debug Link Register
DSPSR_EL0	Debug Saved Program Status Register
DVP RCTX	Data Value Prediction Restriction by Context
ELR_EL1	Exception Link Register (EL1)
ELR_EL2	Exception Link Register (EL2)
ERRIDR_EL1	Error Record ID Register
ERRSELR_EL1	Error Record Select Register
ERXADDR_EL1	Selected Error Record Address Register
ERXCTLR_EL1	Selected Error Record Control Register
ERXFR_EL1	Selected Error Record Feature Register
ERXMISC0_EL1	Selected Error Record Miscellaneous Register 0
ERXMISC1_EL1	Selected Error Record Miscellaneous Register 1
ERXMISC2_EL1	Selected Error Record Miscellaneous Register 2
ERXMISC3_EL1	Selected Error Record Miscellaneous Register 3
ERXPFGCDN_EL1	Selected Pseudo-fault Generation Countdown Register
ERXPFGCTL_EL1	Selected Pseudo-fault Generation Control Register
ERXPFGF_EL1	Selected Pseudo-fault Generation Feature Register
ERXSTATUS_EL1	Selected Error Record Primary Status Register
ESR_EL1	Exception Syndrome Register (EL1)
ESR_EL2	Exception Syndrome Register (EL2)
FAR_EL1	Fault Address Register (EL1)
FAR_EL2	Fault Address Register (EL2)
FPCR	Floating-point Control Register
FPSR	Floating-point Status Register
HACR_EL2	Hypervisor Auxiliary Control Register
HCR_EL2	Hypervisor Configuration Register
HPFAR_EL2	Hypervisor IPA Fault Address Register
HSTR_EL2	Hypervisor System Trap Register
IC IALLU	Instruction Cache Invalidate All to PoU
IC IALLUIS	Instruction Cache Invalidate All to PoU, Inner Shareable
IC IVAU	Instruction Cache line Invalidate by VA to PoU
ICC_ASGI1R_EL1	Interrupt Controller Alias Software Generated Interrupt Group 1 Register
ICC_BPR0_EL1	Interrupt Controller Binary Point Register 0
ICC_BPR1_EL1	Interrupt Controller Binary Point Register 1

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
ICC_CTLR_EL1	Interrupt Controller Control Register (EL1)
ICC_DIR_EL1	Interrupt Controller Deactivate Interrupt Register
ICC_EOIR0_EL1	Interrupt Controller End Of Interrupt Register 0
ICC_EOIR1_EL1	Interrupt Controller End Of Interrupt Register 1
ICC_HPPIR0_EL1	Interrupt Controller Highest Priority Pending Interrupt Register 0
ICC_HPPIR1_EL1	Interrupt Controller Highest Priority Pending Interrupt Register 1
ICC_IAR0_EL1	Interrupt Controller Interrupt Acknowledge Register 0
ICC_IAR1_EL1	Interrupt Controller Interrupt Acknowledge Register 1
ICC_IGRPEN0_EL1	Interrupt Controller Interrupt Group 0 Enable Register
ICC_IGRPEN1_EL1	Interrupt Controller Interrupt Group 1 Enable Register
ICC_PMR_EL1	Interrupt Controller Interrupt Priority Mask Register
ICC_RPR_EL1	Interrupt Controller Running Priority Register
ICC_SGI0R_EL1	Interrupt Controller Software Generated Interrupt Group 0 Register
ICC_SGI1R_EL1	Interrupt Controller Software Generated Interrupt Group 1 Register
ICC_SRE_EL1	Interrupt Controller System Register Enable Register (EL1)
ICC_SRE_EL2	Interrupt Controller System Register Enable Register (EL2)
ICH_EISR_EL2	Interrupt Controller End of Interrupt Status Register
ICH_ELRSR_EL2	Interrupt Controller Empty List Register Status Register
ICH_HCR_EL2	Interrupt Controller Hyp Control Register
ICH_MISR_EL2	Interrupt Controller Maintenance Interrupt State Register
ICH_VMCR_EL2	Interrupt Controller Virtual Machine Control Register
ICH_VTR_EL2	Interrupt Controller VGIC Type Register
ICV_BPR0_EL1	Interrupt Controller Virtual Binary Point Register 0
ICV_BPR1_EL1	Interrupt Controller Virtual Binary Point Register 1
ICV_CTLR_EL1	Interrupt Controller Virtual Control Register
ICV_DIR_EL1	Interrupt Controller Deactivate Virtual Interrupt Register
ICV_EOIR0_EL1	Interrupt Controller Virtual End Of Interrupt Register 0
ICV_EOIR1_EL1	Interrupt Controller Virtual End Of Interrupt Register 1
ICV_HPPIR0_EL1	Interrupt Controller Virtual Highest Priority Pending Interrupt Register 0
ICV_HPPIR1_EL1	Interrupt Controller Virtual Highest Priority Pending Interrupt Register 1
ICV_IAR0_EL1	Interrupt Controller Virtual Interrupt Acknowledge Register 0
ICV_IAR1_EL1	Interrupt Controller Virtual Interrupt Acknowledge Register 1
ICV_IGRPEN0_EL1	Interrupt Controller Virtual Interrupt Group 0 Enable Register
ICV_IGRPEN1_EL1	Interrupt Controller Virtual Interrupt Group 1 Enable Register
ICV_PMR_EL1	Interrupt Controller Virtual Interrupt Priority Mask Register
ICV_RPR_EL1	Interrupt Controller Virtual Running Priority Register
ID_AA64AFR0_EL1	AArch64 Auxiliary Feature Register 0
ID_AA64AFR1_EL1	AArch64 Auxiliary Feature Register 1
ID_AA64DFR0_EL1	AArch64 Debug Feature Register 0

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
ID_AA64DFR1_EL1	AArch64 Debug Feature Register 1
ID_AA64DFR2_EL1	AArch64 Debug Feature Register 2
ID_AA64ISAR0_EL1	AArch64 Instruction Set Attribute Register 0
ID_AA64ISAR1_EL1	AArch64 Instruction Set Attribute Register 1
ID_AA64ISAR2_EL1	AArch64 Instruction Set Attribute Register 2
ID_AA64MMFR0_EL1	AArch64 Memory Model Feature Register 0
ID_AA64MMFR1_EL1	AArch64 Memory Model Feature Register 1
ID_AA64MMFR2_EL1	AArch64 Memory Model Feature Register 2
ID_AA64MMFR3_EL1	AArch64 Memory Model Feature Register 3
ID_AA64MMFR4_EL1	AArch64 Memory Model Feature Register 4
ID_AA64PFR0_EL1	AArch64 Processor Feature Register 0
ID_AA64PFR1_EL1	AArch64 Processor Feature Register 1
ID_AA64PFR2_EL1	AArch64 Processor Feature Register 2
ID_AFR0_EL1	AArch32 Auxiliary Feature Register 0
ID_DFR0_EL1	AArch32 Debug Feature Register 0
ID_DFR1_EL1	AArch32 Debug Feature Register 1
ID_ISAR0_EL1	AArch32 Instruction Set Attribute Register 0
ID_ISAR1_EL1	AArch32 Instruction Set Attribute Register 1
ID_ISAR2_EL1	AArch32 Instruction Set Attribute Register 2
ID_ISAR3_EL1	AArch32 Instruction Set Attribute Register 3
ID_ISAR4_EL1	AArch32 Instruction Set Attribute Register 4
ID_ISAR5_EL1	AArch32 Instruction Set Attribute Register 5
ID_ISAR6_EL1	AArch32 Instruction Set Attribute Register 6
ID_MMFR0_EL1	AArch32 Memory Model Feature Register 0
ID_MMFR1_EL1	AArch32 Memory Model Feature Register 1
ID_MMFR2_EL1	AArch32 Memory Model Feature Register 2
ID_MMFR3_EL1	AArch32 Memory Model Feature Register 3
ID_MMFR4_EL1	AArch32 Memory Model Feature Register 4
ID_MMFR5_EL1	AArch32 Memory Model Feature Register 5
ID_PFR0_EL1	AArch32 Processor Feature Register 0
ID_PFR1_EL1	AArch32 Processor Feature Register 1
ID_PFR2_EL1	AArch32 Processor Feature Register 2
ISR_EL1	Interrupt Status Register
MAIR_EL1	Memory Attribute Indirection Register (EL1)
MAIR_EL2	Memory Attribute Indirection Register (EL2)
MDCCINT_EL1	Monitor DCC Interrupt Enable Register
MDCCSR_EL0	Monitor DCC Status Register
MDCR_EL2	Monitor Debug Configuration Register (EL2)
MDRAR_EL1	Monitor Debug ROM Address Register
MDSCR_EL1	Monitor Debug System Control Register

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
MIDR_EL1	Main ID Register
MPIDR_EL1	Multiprocessor Affinity Register
MPUIR_EL1	MPU Type Register (EL1)
MPUIR_EL2	MPU Type Register (EL2)
MVFR0_EL1	AArch32 Media and VFP Feature Register 0
MVFR1_EL1	AArch32 Media and VFP Feature Register 1
MVFR2_EL1	AArch32 Media and VFP Feature Register 2
NZCV	Condition Flags
OSDLR_EL1	OS Double Lock Register
OSDTRRX_EL1	OS Lock Data Transfer Register, Receive
OSDTRTX_EL1	OS Lock Data Transfer Register, Transmit
OSECCR_EL1	OS Lock Exception Catch Control Register
OSLAR_EL1	OS Lock Access Register
OSLSR_EL1	OS Lock Status Register
PAN	Privileged Access Never
PAR_EL1	Physical Address Register
PMCCFILTR_EL0	Performance Monitors Cycle Count Filter Register
PMCCNTR_EL0	Performance Monitors Cycle Count Register
PMCEID0_EL0	Performance Monitors Common Event Identification Register 0
PMCEID1_EL0	Performance Monitors Common Event Identification Register 1
PMCNTENCLR_EL0	Performance Monitors Count Enable Clear Register
PMCNTENSET_EL0	Performance Monitors Count Enable Set Register
PMCR_EL0	Performance Monitors Control Register
PMINTENCLR_EL1	Performance Monitors Interrupt Enable Clear Register
PMINTENSET_EL1	Performance Monitors Interrupt Enable Set Register
PMMIR_EL1	Performance Monitors Machine Identification Register
PMOVSCLR_EL0	Performance Monitors Overflow Flag Status Clear Register
PMOVSSET_EL0	Performance Monitors Overflow Flag Status Set Register
PMSELR_EL0	Performance Monitors Event Counter Selection Register
PMSWINC_EL0	Performance Monitors Software Increment Register
PMUSERENR_EL0	Performance Monitors User Enable Register
PMXEVCNTR_EL0	Performance Monitors Selected Event Count Register
PMXEVTYPER_EL0	Performance Monitors Selected Event Type Register
PRBAR_EL1	Protection Region Base Address Register (EL1)
PRBAR_EL2	Protection Region Base Address Register (EL2)
PRENR_EL1	Protection Region Enable Register (EL1)
PRENR_EL2	Protection Region Enable Register (EL2)
PRLAR_EL1	Protection Region Limit Address Register (EL1)
PRLAR_EL2	Protection Region Limit Address Register (EL2)
PRSELR_EL1	Protection Region Selection Register (EL1)

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
PRSELR_EL2	Protection Region Selection Register (EL2)
REVIDR_EL1	Revision ID Register
RMR_EL2	Reset Management Register (EL2)
RVBAR_EL2	Reset Vector Base Address Register (if EL3 not implemented)
S1_<op1><*Cn*><Cm>_<op2>	IMPLEMENTATION DEFINED System instructions
S3_<op1><*Cn*><Cm>_<op2>	IMPLEMENTATION DEFINED Registers
SCTLR_EL1	System Control Register (EL1)
SCTLR_EL2	System Control Register (EL2)
SCXTNUM_EL0	EL0 Read-Write Software Context Number
SCXTNUM_EL1	EL1 Read-Write Software Context Number
SCXTNUM_EL2	EL2 Read-Write Software Context Number
SP_EL0	Stack Pointer (EL0)
SP_EL1	Stack Pointer (EL1)
SP_EL2	Stack Pointer (EL2)
SPSel	Stack Pointer Select
SPSR_abt	Saved Program Status Register (Abort mode)
SPSR_EL1	Saved Program Status Register (EL1)
SPSR_EL2	Saved Program Status Register (EL2)
SPSR_fiq	Saved Program Status Register (FIQ mode)
SPSR_irq	Saved Program Status Register (IRQ mode)
SPSR_und	Saved Program Status Register (Undefined mode)
SSBS	Speculative Store Bypass Safe
TCR_EL1	Translation Control Register (EL1)
TCR_EL2	Translation Control Register (EL2)
TLBI ALLE1	TLB Invalidate All, EL1
TLBI ALLE1IS	TLB Invalidate All, EL1, Inner Shareable
TLBI ALLE1OS	TLB Invalidate All, EL1, Outer Shareable
TLBI ALLE2	TLB Invalidate All, EL2
TLBI ALLE2IS	TLB Invalidate All, EL2, Inner Shareable
TLBI ALLE2OS	TLB Invalidate All, EL2, Outer Shareable
TLBI ALLE3	TLB Invalidate All, EL3
TLBI ALLE3IS	TLB Invalidate All, EL3, Inner Shareable
TLBI ALLE3OS	TLB Invalidate All, EL3, Outer Shareable
TLBI ASIDE1	TLB Invalidate by ASID, EL1
TLBI ASIDE1IS	TLB Invalidate by ASID, EL1, Inner Shareable
TLBI ASIDE1OS	TLB Invalidate by ASID, EL1, Outer Shareable
TLBI IPAS2E1	TLB Invalidate by Intermediate Physical Address, Stage 2, EL1
TLBI IPAS2E1IS	TLB Invalidate by Intermediate Physical Address, Stage 2, EL1, Inner Shareable
TLBI IPAS2E1OS	TLB Invalidate by Intermediate Physical Address, Stage 2, EL1, Outer Shareable

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
TLBI IPAS2LE1	TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1
TLBI IPAS2LE1IS	TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Inner Shareable
TLBI IPAS2LE1OS	TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Outer Shareable
TLBI RIPAS2E1	TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1
TLBI RIPAS2E1IS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1, Inner Shareable
TLBI RIPAS2E1OS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1, Outer Shareable
TLBI RIPAS2LE1	TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1
TLBI RIPAS2LE1IS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Inner Shareable
TLBI RIPAS2LE1OS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Outer Shareable
TLBI RVAAE1	TLB Range Invalidate by VA, All ASID, EL1
TLBI RVAAE1IS	TLB Range Invalidate by VA, All ASID, EL1, Inner Shareable
TLBI RVAAE1OS	TLB Range Invalidate by VA, All ASID, EL1, Outer Shareable
TLBI RVAALE1	TLB Range Invalidate by VA, All ASID, Last level, EL1
TLBI RVAALE1IS	TLB Range Invalidate by VA, All ASID, Last Level, EL1, Inner Shareable
TLBI RVAALE1OS	TLB Range Invalidate by VA, All ASID, Last Level, EL1, Outer Shareable
TLBI RVAE1	TLB Range Invalidate by VA, EL1
TLBI RVAE1IS	TLB Range Invalidate by VA, EL1, Inner Shareable
TLBI RVAE1OS	TLB Range Invalidate by VA, EL1, Outer Shareable
TLBI RVAE2	TLB Range Invalidate by VA, EL2
TLBI RVAE2IS	TLB Range Invalidate by VA, EL2, Inner Shareable
TLBI RVAE2OS	TLB Range Invalidate by VA, EL2, Outer Shareable
TLBI RVAE3	TLB Range Invalidate by VA, EL3
TLBI RVAE3IS	TLB Range Invalidate by VA, EL3, Inner Shareable
TLBI RVAE3OS	TLB Range Invalidate by VA, EL3, Outer Shareable
TLBI RVALE1	TLB Range Invalidate by VA, Last level, EL1
TLBI RVALE1IS	TLB Range Invalidate by VA, Last level, EL1, Inner Shareable
TLBI RVALE1OS	TLB Range Invalidate by VA, Last level, EL1, Outer Shareable
TLBI RVALE2	TLB Range Invalidate by VA, Last level, EL2
TLBI RVALE2IS	TLB Range Invalidate by VA, Last level, EL2, Inner Shareable
TLBI RVALE2OS	TLB Range Invalidate by VA, Last level, EL2, Outer Shareable
TLBI RVALE3	TLB Range Invalidate by VA, Last level, EL3
TLBI RVALE3IS	TLB Range Invalidate by VA, Last level, EL3, Inner Shareable
TLBI RVALE3OS	TLB Range Invalidate by VA, Last level, EL3, Outer Shareable

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
TLBI VAAE1	TLB Invalidate by VA, All ASID, EL1
TLBI VAAE1IS	TLB Invalidate by VA, All ASID, EL1, Inner Shareable
TLBI VAAE1OS	TLB Invalidate by VA, All ASID, EL1, Outer Shareable
TLBI VAALE1	TLB Invalidate by VA, All ASID, Last level, EL1
TLBI VAALE1IS	TLB Invalidate by VA, All ASID, Last Level, EL1, Inner Shareable
TLBI VAALE1OS	TLB Invalidate by VA, All ASID, Last Level, EL1, Outer Shareable
TLBI VAE1	TLB Invalidate by VA, EL1
TLBI VAE1IS	TLB Invalidate by VA, EL1, Inner Shareable
TLBI VAE1OS	TLB Invalidate by VA, EL1, Outer Shareable
TLBI VAE2	TLB Invalidate by VA, EL2
TLBI VAE2IS	TLB Invalidate by VA, EL2, Inner Shareable
TLBI VAE2OS	TLB Invalidate by VA, EL2, Outer Shareable
TLBI VAE3	TLB Invalidate by VA, EL3
TLBI VAE3IS	TLB Invalidate by VA, EL3, Inner Shareable
TLBI VAE3OS	TLB Invalidate by VA, EL3, Outer Shareable
TLBI VALE1	TLB Invalidate by VA, Last level, EL1
TLBI VALE1IS	TLB Invalidate by VA, Last level, EL1, Inner Shareable
TLBI VALE1OS	TLB Invalidate by VA, Last level, EL1, Outer Shareable
TLBI VALE2	TLB Invalidate by VA, Last level, EL2
TLBI VALE2IS	TLB Invalidate by VA, Last level, EL2, Inner Shareable
TLBI VALE2OS	TLB Invalidate by VA, Last level, EL2, Outer Shareable
TLBI VALE3	TLB Invalidate by VA, Last level, EL3
TLBI VALE3IS	TLB Invalidate by VA, Last level, EL3, Inner Shareable
TLBI VALE3OS	TLB Invalidate by VA, Last level, EL3, Outer Shareable
TLBI VMALLE1	TLB Invalidate by VMID, All at stage 1, EL1
TLBI VMALLE1IS	TLB Invalidate by VMID, All at stage 1, EL1, Inner Shareable
TLBI VMALLE1OS	TLB Invalidate by VMID, All at stage 1, EL1, Outer Shareable
TLBI VMALLS12E1	TLB Invalidate by VMID, All at Stage 1 and 2, EL1
TLBI VMALLS12E1IS	TLB Invalidate by VMID, All at Stage 1 and 2, EL1, Inner Shareable
TLBI VMALLS12E1OS	TLB Invalidate by VMID, All at Stage 1 and 2, EL1, Outer Shareable
TPIDR_EL0	EL0 Read-Write Software Thread ID Register
TPIDR_EL1	EL1 Software Thread ID Register
TPIDR_EL2	EL2 Software Thread ID Register
TPIDRRO_EL0	EL0 Read-Only Software Thread ID Register
TRFCR_EL1	Trace Filter Control Register (EL1)
TRFCR_EL2	Trace Filter Control Register (EL2)
TTBR0_EL1	Translation Table Base Register 0 (EL1)
TTBR1_EL1	Translation Table Base Register 1 (EL1)
UAO	User Access Override
VBAR_EL1	Vector Base Address Register (EL1)

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
VBAR_EL2	Vector Base Address Register (EL2)
VDISR_EL2	Virtual Deferred Interrupt Status Register (EL2)
VMPIDR_EL2	Virtualization Multiprocessor ID Register
VPIDR_EL2	Virtualization Processor ID Register
VSCTLR_EL2	Virtualization System Control Register (EL2)
VSESR_EL2	Virtual SError Exception Syndrome Register (EL2)
VSTCR_EL2	Virtualization Secure Translation Control Register
VTCR_EL2	Virtualization Translation Control Register

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H4.3 Functional index of AArch64 registers and System instructions

This section is an index of the AArch64 registers and System instructions, divided by functional group. Each of the following sections lists the registers for a functional group:

- Special-purpose registers.
- VMSA-specific registers.
- ID registers.
- Performance monitors registers.
- Debug registers.
- Trace registers.
- RAS registers.
- Generic timer registers.
- Cache maintenance system instructions.
- Address translation system instructions.
- TLB maintenance system instructions.
- Prediction restriction System instructions.
- Base system registers.

H4.3.1 Special-purpose registers

This section is an index to the registers in the Special-purpose registers functional group.

Table H4-3 Special-purpose registers

Register	Description, see
DLR_EL0	Debug Link Register
DSPSR_EL0	Debug Saved Program Status Register
ELR_EL1	Exception Link Register (EL1)
ELR_EL2	Exception Link Register (EL2)
SP_EL0	Stack Pointer (EL0)
SP_EL1	Stack Pointer (EL1)
SP_EL2	Stack Pointer (EL2)
SPSR_abt	Saved Program Status Register (Abort mode)
SPSR_EL1	Saved Program Status Register (EL1)
SPSR_EL2	Saved Program Status Register (EL2)
SPSR_fiq	Saved Program Status Register (FIQ mode)
SPSR_irq	Saved Program Status Register (IRQ mode)
SPSR_und	Saved Program Status Register (Undefined mode)

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H4.3.2 VMSA-specific registers

This section is an index to the registers in the Virtual memory control registers functional group.

Table H4-4 VMSA-specific registers

Register	Description, see
AMAIR_EL1	Auxiliary Memory Attribute Indirection Register (EL1)
AMAIR_EL2	Auxiliary Memory Attribute Indirection Register (EL2)
CONTEXTIDR_EL1	Context ID Register (EL1)
CONTEXTIDR_EL2	Context ID Register (EL2)
MAIR_EL1	Memory Attribute Indirection Register (EL1)
MAIR_EL2	Memory Attribute Indirection Register (EL2)
TCR_EL1	Translation Control Register (EL1)
TCR_EL2	Translation Control Register (EL2)
TTBR0_EL1	Translation Table Base Register 0 (EL1)
TTBR1_EL1	Translation Table Base Register 1 (EL1)
VTCR_EL2	Virtualization Translation Control Register

H4.3.3 ID registers

This section is an index to the registers in the Identification registers functional group.

Table H4-5 ID registers

Register	Description, see
CCSIDR2_EL1	Current Cache Size ID Register 2
CCSIDR_EL1	Current Cache Size ID Register
CLIDR_EL1	Cache Level ID Register
CSSELR_EL1	Cache Size Selection Register
CTR_EL0	Cache Type Register
DCZID_EL0	Data Cache Zero ID Register
ID_AA64AFR0_EL1	AArch64 Auxiliary Feature Register 0
ID_AA64AFR1_EL1	AArch64 Auxiliary Feature Register 1
ID_AA64DFR0_EL1	AArch64 Debug Feature Register 0
ID_AA64DFR1_EL1	AArch64 Debug Feature Register 1
ID_AA64DFR2_EL1	AArch64 Debug Feature Register 2
ID_AA64ISAR0_EL1	AArch64 Instruction Set Attribute Register 0
ID_AA64ISAR1_EL1	AArch64 Instruction Set Attribute Register 1
ID_AA64ISAR2_EL1	AArch64 Instruction Set Attribute Register 2
ID_AA64MMFR0_EL1	AArch64 Memory Model Feature Register 0
ID_AA64MMFR1_EL1	AArch64 Memory Model Feature Register 1

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
ID_AA64MMFR2_EL1	AArch64 Memory Model Feature Register 2
ID_AA64MMFR3_EL1	AArch64 Memory Model Feature Register 3
ID_AA64MMFR4_EL1	AArch64 Memory Model Feature Register 4
ID_AA64PFR0_EL1	AArch64 Processor Feature Register 0
ID_AA64PFR1_EL1	AArch64 Processor Feature Register 1
ID_AA64PFR2_EL1	AArch64 Processor Feature Register 2
ID_AFR0_EL1	AArch32 Auxiliary Feature Register 0
ID_DFR0_EL1	AArch32 Debug Feature Register 0
ID_DFR1_EL1	AArch32 Debug Feature Register 1
ID_ISAR0_EL1	AArch32 Instruction Set Attribute Register 0
ID_ISAR1_EL1	AArch32 Instruction Set Attribute Register 1
ID_ISAR2_EL1	AArch32 Instruction Set Attribute Register 2
ID_ISAR3_EL1	AArch32 Instruction Set Attribute Register 3
ID_ISAR4_EL1	AArch32 Instruction Set Attribute Register 4
ID_ISAR5_EL1	AArch32 Instruction Set Attribute Register 5
ID_ISAR6_EL1	AArch32 Instruction Set Attribute Register 6
ID_MMFR0_EL1	AArch32 Memory Model Feature Register 0
ID_MMFR1_EL1	AArch32 Memory Model Feature Register 1
ID_MMFR2_EL1	AArch32 Memory Model Feature Register 2
ID_MMFR3_EL1	AArch32 Memory Model Feature Register 3
ID_MMFR4_EL1	AArch32 Memory Model Feature Register 4
ID_MMFR5_EL1	AArch32 Memory Model Feature Register 5
ID_PFR0_EL1	AArch32 Processor Feature Register 0
ID_PFR1_EL1	AArch32 Processor Feature Register 1
ID_PFR2_EL1	AArch32 Processor Feature Register 2
MIDR_EL1	Main ID Register
MPIDR_EL1	Multiprocessor Affinity Register
MVFR0_EL1	AArch32 Media and VFP Feature Register 0
MVFR1_EL1	AArch32 Media and VFP Feature Register 1
MVFR2_EL1	AArch32 Media and VFP Feature Register 2
REVIDR_EL1	Revision ID Register
VMPIDR_EL2	Virtualization Multiprocessor ID Register
VPIDR_EL2	Virtualization Processor ID Register

H4.3.4 Performance monitors registers

This section is an index to the registers in the Performance Monitors registers functional group.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table H4-6 Performance monitors registers

Register	Description, see
PMCCFILTR_EL0	Performance Monitors Cycle Count Filter Register
PMCCNTR_EL0	Performance Monitors Cycle Count Register
PMCEID0_EL0	Performance Monitors Common Event Identification Register 0
PMCEID1_EL0	Performance Monitors Common Event Identification Register 1
PMCNTENCLR_EL0	Performance Monitors Count Enable Clear Register
PMCNTENSET_EL0	Performance Monitors Count Enable Set Register
PMCR_EL0	Performance Monitors Control Register
PMINTENCLR_EL1	Performance Monitors Interrupt Enable Clear Register
PMINTENSET_EL1	Performance Monitors Interrupt Enable Set Register
PMMIR_EL1	Performance Monitors Machine Identification Register
PMOVSCLR_EL0	Performance Monitors Overflow Flag Status Clear Register
PMOVSSET_EL0	Performance Monitors Overflow Flag Status Set Register
PMSELR_EL0	Performance Monitors Event Counter Selection Register
PMSWINC_EL0	Performance Monitors Software Increment Register
PMUSERENR_EL0	Performance Monitors User Enable Register
PMXEVCNTR_EL0	Performance Monitors Selected Event Count Register
PMXEVTYPER_EL0	Performance Monitors Selected Event Type Register

H4.3.5 Debug registers

This section is an index to the registers in the Debug registers functional group.

Table H4-7 Debug registers

Register	Description, see
DBGAUTHSTATUS_EL1	Debug Authentication Status Register
DBGCLAIMCLR_EL1	Debug CLAIM Tag Clear Register
DBGCLAIMSET_EL1	Debug CLAIM Tag Set Register
DBGDTR_EL0	Debug Data Transfer Register, half-duplex
DBGDTRRX_EL0	Debug Data Transfer Register, Receive
DBGDTRTX_EL0	Debug Data Transfer Register, Transmit
DBGPRCR_EL1	Debug Power Control Register
MDCCINT_EL1	Monitor DCC Interrupt Enable Register
MDCCSR_EL0	Monitor DCC Status Register
MDCR_EL2	Monitor Debug Configuration Register (EL2)
MDRAR_EL1	Monitor Debug ROM Address Register

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
MDSCR_EL1	Monitor Debug System Control Register
OSDLR_EL1	OS Double Lock Register
OSDTRRX_EL1	OS Lock Data Transfer Register, Receive
OSDTRTX_EL1	OS Lock Data Transfer Register, Transmit
OSECCR_EL1	OS Lock Exception Catch Control Register
OSLAR_EL1	OS Lock Access Register
OSLSR_EL1	OS Lock Status Register
TRFCR_EL1	Trace Filter Control Register (EL1)
TRFCR_EL2	Trace Filter Control Register (EL2)

H4.3.6 Trace registers

This section is an index to the registers in the Trace unit registers functional group.

Table H4-8 Trace registers

Description, see

H4.3.7 RAS registers

This section is an index to the registers in the RAS registers functional group.

Table H4-9 RAS registers

Register	Description, see
DISR_EL1	Deferred Interrupt Status Register
ERRIDR_EL1	Error Record ID Register
ERRSELR_EL1	Error Record Select Register
ERXADDR_EL1	Selected Error Record Address Register
ERXCTLR_EL1	Selected Error Record Control Register
ERXFR_EL1	Selected Error Record Feature Register
ERXMISC0_EL1	Selected Error Record Miscellaneous Register 0
ERXMISC1_EL1	Selected Error Record Miscellaneous Register 1
ERXMISC2_EL1	Selected Error Record Miscellaneous Register 2
ERXMISC3_EL1	Selected Error Record Miscellaneous Register 3
ERXPFGCDN_EL1	Selected Pseudo-fault Generation Countdown Register
ERXPFGCTL_EL1	Selected Pseudo-fault Generation Control Register
ERXPFGF_EL1	Selected Pseudo-fault Generation Feature Register

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
ERXSTATUS_EL1	Selected Error Record Primary Status Register
VDISR_EL2	Virtual Deferred Interrupt Status Register (EL2)
VSESR_EL2	Virtual SError Exception Syndrome Register (EL2)

H4.3.8 Generic timer registers

This section is an index to the registers in the Generic Timer registers functional group.

Table H4-10 Generic timer registers

Register	Description, see
CNTFRQ_EL0	Counter-timer Frequency Register
CNTHCTL_EL2	Counter-timer Hypervisor Control Register
CNTHPS_CTL_EL2	Counter-timer Secure Physical Timer Control Register (EL2)
CNTHPS_CVAL_EL2	Counter-timer Secure Physical Timer CompareValue Register (EL2)
CNTHPS_TVAL_EL2	Counter-timer Secure Physical Timer TimerValue Register (EL2)
CNTKCTL_EL1	Counter-timer Kernel Control Register
CNTP_CTL_EL0	Counter-timer Physical Timer Control Register
CNTP_CVAL_EL0	Counter-timer Physical Timer CompareValue Register
CNTP_TVAL_EL0	Counter-timer Physical Timer TimerValue Register
CNTPCT_EL0	Counter-timer Physical Count Register
CNTV_CTL_EL0	Counter-timer Virtual Timer Control Register
CNTV_CVAL_EL0	Counter-timer Virtual Timer CompareValue Register
CNTV_TVAL_EL0	Counter-timer Virtual Timer TimerValue Register
CNTVCT_EL0	Counter-timer Virtual Count Register
CNTVOFF_EL2	Counter-timer Virtual Offset Register

H4.3.9 Cache maintenance system instructions

This section is an index to the registers in the Cache maintenance instructions functional group.

Table H4-11 Cache maintenance system instructions

System instruction	Description, see
DC CISW	Data or unified Cache line Clean and Invalidate by Set-Way
DC CIVAC	Data or unified Cache line Clean and Invalidate by VA to PoC
DC CSW	Data or unified Cache line Clean by Set-Way
DC CVAC	Data or unified Cache line Clean by VA to PoC

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

System instruction	Description, see
DC CVADP	Data or unified Cache line Clean by VA to PoDP
DC CVAP	Data or unified Cache line Clean by VA to PoP
DC CVAU	Data or unified Cache line Clean by VA to PoU
DC ISW	Data or unified Cache line Invalidate by Set-Way
DC IVAC	Data or unified Cache line Invalidate by VA to PoC
DC ZVA	Data Cache Zero by VA
IC IALLU	Instruction Cache Invalidate All to PoU
IC IALLUIS	Instruction Cache Invalidate All to PoU, Inner Shareable
IC IVAU	Instruction Cache line Invalidate by VA to PoU

H4.3.10 Address translation system instructions

This section is an index to the registers in the Address translation instructions functional group.

Table H4-12 Address translation system instructions

System instruction	Description, see
AT S12E0R	Address Translate Stages 1 and 2 EL0 Read
AT S12E0W	Address Translate Stages 1 and 2 EL0 Write
AT S12E1R	Address Translate Stages 1 and 2 EL1 Read
AT S12E1W	Address Translate Stages 1 and 2 EL1 Write
AT S1E0R	Address Translate Stage 1 EL0 Read
AT S1E0W	Address Translate Stage 1 EL0 Write
AT S1E1R	Address Translate Stage 1 EL1 Read
AT S1E1RP	Address Translate Stage 1 EL1 Read PAN
AT S1E1W	Address Translate Stage 1 EL1 Write
AT S1E1WP	Address Translate Stage 1 EL1 Write PAN
AT S1E2R	Address Translate Stage 1 EL2 Read
AT S1E2W	Address Translate Stage 1 EL2 Write
AT S1E3R	Address Translate Stage 1 EL3 Read
AT S1E3W	Address Translate Stage 1 EL3 Write

H4.3.11 TLB maintenance system instructions

This section is an index to the registers in the TLB maintenance instructions functional group.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table H4-13 TLB maintenance system instructions

System instruction	Description, see
TLBI ALLE1	TLB Invalidate All, EL1
TLBI ALLE1IS	TLB Invalidate All, EL1, Inner Shareable
TLBI ALLE1OS	TLB Invalidate All, EL1, Outer Shareable
TLBI ALLE2	TLB Invalidate All, EL2
TLBI ALLE2IS	TLB Invalidate All, EL2, Inner Shareable
TLBI ALLE2OS	TLB Invalidate All, EL2, Outer Shareable
TLBI ALLE3	TLB Invalidate All, EL3
TLBI ALLE3IS	TLB Invalidate All, EL3, Inner Shareable
TLBI ALLE3OS	TLB Invalidate All, EL3, Outer Shareable
TLBI ASIDE1	TLB Invalidate by ASID, EL1
TLBI ASIDE1IS	TLB Invalidate by ASID, EL1, Inner Shareable
TLBI ASIDE1OS	TLB Invalidate by ASID, EL1, Outer Shareable
TLBI IPAS2E1	TLB Invalidate by Intermediate Physical Address, Stage 2, EL1
TLBI IPAS2E1IS	TLB Invalidate by Intermediate Physical Address, Stage 2, EL1, Inner Shareable
TLBI IPAS2E1OS	TLB Invalidate by Intermediate Physical Address, Stage 2, EL1, Outer Shareable
TLBI IPAS2LE1	TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1
TLBI IPAS2LE1IS	TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Inner Shareable
TLBI IPAS2LE1OS	TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Outer Shareable
TLBI RIPAS2E1	TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1
TLBI RIPAS2E1IS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1, Inner Shareable
TLBI RIPAS2E1OS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1, Outer Shareable
TLBI RIPAS2LE1	TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1
TLBI RIPAS2LE1IS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Inner Shareable
TLBI RIPAS2LE1OS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Outer Shareable
TLBI RVAAE1	TLB Range Invalidate by VA, All ASID, EL1
TLBI RVAAE1IS	TLB Range Invalidate by VA, All ASID, EL1, Inner Shareable
TLBI RVAAE1OS	TLB Range Invalidate by VA, All ASID, EL1, Outer Shareable
TLBI RVAALE1	TLB Range Invalidate by VA, All ASID, Last level, EL1
TLBI RVAALE1IS	TLB Range Invalidate by VA, All ASID, Last Level, EL1, Inner Shareable
TLBI RVAALE1OS	TLB Range Invalidate by VA, All ASID, Last Level, EL1, Outer Shareable

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

System instruction	Description, see
TLBI RVAE1	TLB Range Invalidate by VA, EL1
TLBI RVAE1IS	TLB Range Invalidate by VA, EL1, Inner Shareable
TLBI RVAE1OS	TLB Range Invalidate by VA, EL1, Outer Shareable
TLBI RVAE2	TLB Range Invalidate by VA, EL2
TLBI RVAE2IS	TLB Range Invalidate by VA, EL2, Inner Shareable
TLBI RVAE2OS	TLB Range Invalidate by VA, EL2, Outer Shareable
TLBI RVAE3	TLB Range Invalidate by VA, EL3
TLBI RVAE3IS	TLB Range Invalidate by VA, EL3, Inner Shareable
TLBI RVAE3OS	TLB Range Invalidate by VA, EL3, Outer Shareable
TLBI RVALE1	TLB Range Invalidate by VA, Last level, EL1
TLBI RVALE1IS	TLB Range Invalidate by VA, Last level, EL1, Inner Shareable
TLBI RVALE1OS	TLB Range Invalidate by VA, Last level, EL1, Outer Shareable
TLBI RVALE2	TLB Range Invalidate by VA, Last level, EL2
TLBI RVALE2IS	TLB Range Invalidate by VA, Last level, EL2, Inner Shareable
TLBI RVALE2OS	TLB Range Invalidate by VA, Last level, EL2, Outer Shareable
TLBI RVALE3	TLB Range Invalidate by VA, Last level, EL3
TLBI RVALE3IS	TLB Range Invalidate by VA, Last level, EL3, Inner Shareable
TLBI RVALE3OS	TLB Range Invalidate by VA, Last level, EL3, Outer Shareable
TLBI VAAE1	TLB Invalidate by VA, All ASID, EL1
TLBI VAAE1IS	TLB Invalidate by VA, All ASID, EL1, Inner Shareable
TLBI VAAE1OS	TLB Invalidate by VA, All ASID, EL1, Outer Shareable
TLBI VAALE1	TLB Invalidate by VA, All ASID, Last level, EL1
TLBI VAALE1IS	TLB Invalidate by VA, All ASID, Last Level, EL1, Inner Shareable
TLBI VAALE1OS	TLB Invalidate by VA, All ASID, Last Level, EL1, Outer Shareable
TLBI VAE1	TLB Invalidate by VA, EL1
TLBI VAE1IS	TLB Invalidate by VA, EL1, Inner Shareable
TLBI VAE1OS	TLB Invalidate by VA, EL1, Outer Shareable
TLBI VAE2	TLB Invalidate by VA, EL2
TLBI VAE2IS	TLB Invalidate by VA, EL2, Inner Shareable
TLBI VAE2OS	TLB Invalidate by VA, EL2, Outer Shareable
TLBI VAE3	TLB Invalidate by VA, EL3
TLBI VAE3IS	TLB Invalidate by VA, EL3, Inner Shareable
TLBI VAE3OS	TLB Invalidate by VA, EL3, Outer Shareable
TLBI VALE1	TLB Invalidate by VA, Last level, EL1
TLBI VALE1IS	TLB Invalidate by VA, Last level, EL1, Inner Shareable
TLBI VALE1OS	TLB Invalidate by VA, Last level, EL1, Outer Shareable
TLBI VALE2	TLB Invalidate by VA, Last level, EL2
TLBI VALE2IS	TLB Invalidate by VA, Last level, EL2, Inner Shareable
TLBI VALE2OS	TLB Invalidate by VA, Last level, EL2, Outer Shareable

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

System instruction	Description, see
TLBI VALE3	TLB Invalidate by VA, Last level, EL3
TLBI VALE3IS	TLB Invalidate by VA, Last level, EL3, Inner Shareable
TLBI VALE3OS	TLB Invalidate by VA, Last level, EL3, Outer Shareable
TLBI VMALLE1	TLB Invalidate by VMID, All at stage 1, EL1
TLBI VMALLE1IS	TLB Invalidate by VMID, All at stage 1, EL1, Inner Shareable
TLBI VMALLE1OS	TLB Invalidate by VMID, All at stage 1, EL1, Outer Shareable
TLBI VMALLS12E1	TLB Invalidate by VMID, All at Stage 1 and 2, EL1
TLBI VMALLS12E1IS	TLB Invalidate by VMID, All at Stage 1 and 2, EL1, Inner Shareable
TLBI VMALLS12E1OS	TLB Invalidate by VMID, All at Stage 1 and 2, EL1, Outer Shareable

H4.3.12 Prediction restriction System instructions

This section is an index to the registers in the Prediction restriction instructions functional group.

Table H4-14 Cache maintenance system instructions

System instruction	Description, see
CFP RCTX	Control Flow Prediction Restriction by Context
CPP RCTX	Cache Prefetch Prediction Restriction by Context
DVP RCTX	Data Value Prediction Restriction by Context

H4.3.13 Base system registers

This section is an index to the registers in the functional group.

Table H4-15 Base system registers

Register	Description, see
CPACR_EL1	Architectural Feature Access Control Register
SCTLR_EL1	System Control Register (EL1)
SCTLR_EL2	System Control Register (EL2)

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H4.4 Alphabetical index of memory-mapped registers

This section is an index of memory-mapped registers in alphabetical order.

Table H4-16 Alphabetical index of Memory-Mapped Registers

Register	Description, see
ASICCTL	CTI External Multiplexer Control register
CNTCR	Counter Control Register
CNTCV	Counter Count Value register
CNTEL0ACR	Counter-timer EL0 Access Control Register
CNTFID0	Counter Frequency ID
CNTFRQ	Counter-timer Frequency
CNTID	Counter Identification Register
CNTNSAR	Counter-timer Non-secure Access Register
CNTP_CTL	Counter-timer Physical Timer Control
CNTP_CVAL	Counter-timer Physical Timer CompareValue
CNTP_TVAL	Counter-timer Physical Timer TimerValue
CNTPCT	Counter-timer Physical Count
CNTSCR	Counter Scale Register
CNTSR	Counter Status Register
CNTTIDR	Counter-timer Timer ID Register
CNTV_CTL	Counter-timer Virtual Timer Control
CNTV_CVAL	Counter-timer Virtual Timer CompareValue
CNTV_TVAL	Counter-timer Virtual Timer TimerValue
CNTVCT	Counter-timer Virtual Count
CNTVOFF	Counter-timer Virtual Offset
CTIAPPCLEAR	CTI Application Trigger Clear register
CTIAPPPULSE	CTI Application Pulse register
CTIAPPSET	CTI Application Trigger Set register
CTIAUTHSTATUS	CTI Authentication Status register
CTICHINSTATUS	CTI Channel In Status register
CTICHOUTSTATUS	CTI Channel Out Status register
CTICIDR0	CTI Component Identification Register 0
CTICIDR1	CTI Component Identification Register 1
CTICIDR2	CTI Component Identification Register 2
CTICIDR3	CTI Component Identification Register 3
CTICLAIMCLR	CTI CLAIM Tag Clear register
CTICLAIMSET	CTI CLAIM Tag Set register
CTICONTROL	CTI Control register
CTIDEVAFF0	CTI Device Affinity register 0
CTIDEVAFF1	CTI Device Affinity register 1
CTIDEVARCH	CTI Device Architecture register

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
CTIDEVCTL	CTI Device Control register
CTIDEVID	CTI Device ID register 0
CTIDEVID1	CTI Device ID register 1
CTIDEVID2	CTI Device ID register 2
CTIDEVTYPE	CTI Device Type register
CTIGATE	CTI Channel Gate Enable register
CTIINTACK	CTI Output Trigger Acknowledge register
CTIITCTRL	CTI Integration mode Control register
CTILAR	CTI Lock Access Register
CTILSR	CTI Lock Status Register
CTIPIDR0	CTI Peripheral Identification Register 0
CTIPIDR1	CTI Peripheral Identification Register 1
CTIPIDR2	CTI Peripheral Identification Register 2
CTIPIDR3	CTI Peripheral Identification Register 3
CTIPIDR4	CTI Peripheral Identification Register 4
CTITRIGINSTATUS	CTI Trigger In Status register
CTITRIGOUTSTATUS	CTI Trigger Out Status register
DBGAUTHSTATUS_EL1	Debug Authentication Status Register
DBGCLAIMCLR_EL1	Debug CLAIM Tag Clear Register
DBGCLAIMSET_EL1	Debug CLAIM Tag Set Register
DBGDTRRX_EL0	Debug Data Transfer Register, Receive
DBGDTRTX_EL0	Debug Data Transfer Register, Transmit
EDAA32PFR	External Debug Auxiliary Processor Feature Register
EDACR	External Debug Auxiliary Control Register
EDCIDR0	External Debug Component Identification Register 0
EDCIDR1	External Debug Component Identification Register 1
EDCIDR2	External Debug Component Identification Register 2
EDCIDR3	External Debug Component Identification Register 3
EDDEVAFF0	External Debug Device Affinity register 0
EDDEVAFF1	External Debug Device Affinity register 1
EDDEVARCH	External Debug Device Architecture Register
EDDEVID	External Debug Device ID register 0
EDDEVID1	External Debug Device ID Register 1
EDDEVID2	External Debug Device ID register 2
EDDEVTYPE	External Debug Device Type register
EDDFR	External Debug Feature Register
EDECCR	External Debug Exception Catch Control Register
EDECR	External Debug Execution Control Register
EDESR	External Debug Event Status Register
EDITCTRL	External Debug Integration mode Control register

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
EDITR	External Debug Instruction Transfer Register
EDLAR	External Debug Lock Access Register
EDLSR	External Debug Lock Status Register
EDPFR	External Debug Processor Feature Register
EDPIDR0	External Debug Peripheral Identification Register 0
EDPIDR1	External Debug Peripheral Identification Register 1
EDPIDR2	External Debug Peripheral Identification Register 2
EDPIDR3	External Debug Peripheral Identification Register 3
EDPIDR4	External Debug Peripheral Identification Register 4
EDPRCR	External Debug Power-Reset Control Register
EDPRSR	External Debug Processor Status Register
EDRCR	External Debug Reserve Control Register
EDSCR	External Debug Status and Control Register
EDWAR	External Debug Watchpoint Address Register
ERRCIDR0	Component Identification Register 0
ERRCIDR1	Component Identification Register 1
ERRCIDR2	Component Identification Register 2
ERRCIDR3	Component Identification Register 3
ERRCRICR0	Critical Error Interrupt Configuration Register 0
ERRCRICR1	Critical Error Interrupt Configuration Register 1
ERRCRICR2	Critical Error Interrupt Configuration Register 2
ERRDEVAFF	Device Affinity Register
ERRDEVARCH	Device Architecture Register
ERRDEVID	Device Configuration Register
ERRERICR0	Error Recovery Interrupt Configuration Register 0
ERRERICR1	Error Recovery Interrupt Configuration Register 1
ERRERICR2	Error Recovery Interrupt Configuration Register 2
ERRFHICR0	Fault Handling Interrupt Configuration Register 0
ERRFHICR1	Fault Handling Interrupt Configuration Register 1
ERRFHICR2	Fault Handling Interrupt Configuration Register 2
ERRIIDR	Implementation Identification Register
ERRIRQSR	Error Interrupt Status Register
ERRPIDR0	Peripheral Identification Register 0
ERRPIDR1	Peripheral Identification Register 1
ERRPIDR2	Peripheral Identification Register 2
ERRPIDR3	Peripheral Identification Register 3
ERRPIDR4	Peripheral Identification Register 4
GICC_ABPR	CPU Interface Aliased Binary Point Register
GICC_AEOIR	CPU Interface Aliased End Of Interrupt Register

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
GICC_AHPPIR	CPU Interface Aliased Highest Priority Pending Interrupt Register
GICC_AIAR	CPU Interface Aliased Interrupt Acknowledge Register
GICC_BPR	CPU Interface Binary Point Register
GICC_CTLR	CPU Interface Control Register
GICC_DIR	CPU Interface Deactivate Interrupt Register
GICC_EOIR	CPU Interface End Of Interrupt Register
GICC_HPPIR	CPU Interface Highest Priority Pending Interrupt Register
GICC_IAR	CPU Interface Interrupt Acknowledge Register
GICC_IIDR	CPU Interface Identification Register
GICC_PMR	CPU Interface Priority Mask Register
GICC_RPR	CPU Interface Running Priority Register
GICC_STATUSR	CPU Interface Status Register
GICD_CLRSPI_NSR	Clear Non-secure SPI Pending Register
GICD_CLRSPI_SR	Clear Secure SPI Pending Register
GICD_CTLR	Distributor Control Register
GICD_IIDR	Distributor Implementer Identification Register
GICD_SETSPI_NSR	Set Non-secure SPI Pending Register
GICD_SETSPI_SR	Set Secure SPI Pending Register
GICD_SGIR	Software Generated Interrupt Register
GICD_STATUSR	Error Reporting Status Register
GICD_TYPER	Interrupt Controller Type Register
GICD_TYPER2	Interrupt Controller Type Register 2
GICH_EISR	End Interrupt Status Register
GICH_ELRSR	Empty List Register Status Register
GICH_HCR	Hypervisor Control Register
GICH_MISR	Maintenance Interrupt Status Register
GICH_VMCR	Virtual Machine Control Register
GICH_VTR	Virtual Type Register
GICM_CLRSPI_NSR	Clear Non-secure SPI Pending Register
GICM_CLRSPI_SR	Clear Secure SPI Pending Register
GICM_IIDR	Distributor Implementer Identification Register
GICM_SETSPI_NSR	Set Non-secure SPI Pending Register
GICM_SETSPI_SR	Set Secure SPI Pending Register
GICM_TYPER	Distributor MSI Type Register
GICR_CLRLPIR	Clear LPI Pending Register
GICR_CTLR	Redistributor Control Register
GICR_ICACTIVER0	Interrupt Clear-Active Register 0
GICR_ICENABLER0	Interrupt Clear-Enable Register 0
GICR_ICFGR0	Interrupt Configuration Register 0

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
GICR_ICFGR1	Interrupt Configuration Register 1
GICR_ICPENDR0	Interrupt Clear-Pending Register 0
GICR_IGROUPR0	Interrupt Group Register 0
GICR_IGRPMODR0	Interrupt Group Modifier Register 0
GICR_IIDR	Redistributor Implementer Identification Register
GICR_INVALLR	Redistributor Invalidate All Register
GICR_INVLPIR	Redistributor Invalidate LPI Register
GICR_ISACTIVER0	Interrupt Set-Active Register 0
GICR_ISENABLER0	Interrupt Set-Enable Register 0
GICR_ISPENDR0	Interrupt Set-Pending Register 0
GICR_MPAMIDR	Report maximum PARTID and PMG Register
GICR_NSACR	Non-secure Access Control Register
GICR_PARTIDR	Set PARTID and PMG Register
GICR_PENDBASER	Redistributor LPI Pending Table Base Address Register
GICR_PROPBASER	Redistributor Properties Base Address Register
GICR_SETLPIR	Set LPI Pending Register
GICR_STATUSR	Error Reporting Status Register
GICR_SYNCR	Redistributor Synchronize Register
GICR_TYPER	Redistributor Type Register
GICR_VPENDBASER	Virtual Redistributor LPI Pending Table Base Address Register
GICR_VPROPBASER	Virtual Redistributor Properties Base Address Register
GICR_VSGIPENDR	Redistributor virtual SGI pending state register
GICR_VSGIR	Redistributor virtual SGI pending state request register
GICR_WAKER	Redistributor Wake Register
GICV_ABPR	Virtual Machine Aliased Binary Point Register
GICV_AEOIR	Virtual Machine Aliased End Of Interrupt Register
GICV_AHPPIR	Virtual Machine Aliased Highest Priority Pending Interrupt Register
GICV_AIAR	Virtual Machine Aliased Interrupt Acknowledge Register
GICV_BPR	Virtual Machine Binary Point Register
GICV_CTLR	Virtual Machine Control Register
GICV_DIR	Virtual Machine Deactivate Interrupt Register
GICV_EOIR	Virtual Machine End Of Interrupt Register
GICV_HPPIR	Virtual Machine Highest Priority Pending Interrupt Register
GICV_IAR	Virtual Machine Interrupt Acknowledge Register
GICV_IIDR	Virtual Machine CPU Interface Identification Register
GICV_PMR	Virtual Machine Priority Mask Register
GICV_RPR	Virtual Machine Running Priority Register
GICV_STATUSR	Virtual Machine Error Reporting Status Register

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
GITS_CBASER	ITS Command Queue Descriptor
GITS_CREADR	ITS Read Register
GITS_CTLR	ITS Control Register
GITS_CWRITER	ITS Write Register
GITS_IIDR	ITS Identification Register
GITS_MPAMIDR	Report maximum PARTID and PMG Register
GITS_MPIDR	Report ITS's affinity.
GITS_PARTIDR	Set PARTID and PMG Register
GITS_SGIR	ITS SGI Register
GITS_STATUSR	ITS Error Reporting Status Register
GITS_TRANSLATER	ITS Translation Register
GITS_TYPER	ITS Type Register
GITS_UMSIR	ITS Unmapped MSI register
MIDR_EL1	Main ID Register
OSLAR_EL1	OS Lock Access Register
PMAUTHSTATUS	Performance Monitors Authentication Status register
PMCCFILTR_EL0	Performance Monitors Cycle Counter Filter Register
PMCCNTR_EL0	Performance Monitors Cycle Counter
PMCEID0	Performance Monitors Common Event Identification register 0
PMCEID1	Performance Monitors Common Event Identification register 1
PMCEID2	Performance Monitors Common Event Identification register 2
PMCEID3	Performance Monitors Common Event Identification register 3
PMCFGR	Performance Monitors Configuration Register
PMCID1SR	CONTEXTIDR_EL1 Sample Register
PMCID2SR	CONTEXTIDR_EL2 Sample Register
PMCIDR0	Performance Monitors Component Identification Register 0
PMCIDR1	Performance Monitors Component Identification Register 1
PMCIDR2	Performance Monitors Component Identification Register 2
PMCIDR3	Performance Monitors Component Identification Register 3
PMCNTENCLR_EL0	Performance Monitors Count Enable Clear Register
PMCNTENSET_EL0	Performance Monitors Count Enable Set Register
PMCR_EL0	Performance Monitors Control Register
PMDEVAFF0	Performance Monitors Device Affinity register 0
PMDEVAFF1	Performance Monitors Device Affinity register 1
PMDEVARCH	Performance Monitors Device Architecture register

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
PMDEVID	Performance Monitors Device ID register
PMDEVTYPE	Performance Monitors Device Type register
PMIIDR	Performance Monitors Implementation Identification Register
PMINTENCLR_EL1	Performance Monitors Interrupt Enable Clear Register
PMINTENSET_EL1	Performance Monitors Interrupt Enable Set Register
PMITCTRL	Performance Monitors Integration mode Control register
PMLAR	Performance Monitors Lock Access Register
PMLSR	Performance Monitors Lock Status Register
PMMIR	Performance Monitors Machine Identification Register
PMOVSCLR_EL0	Performance Monitors Overflow Flag Status Clear register
PMOVSSET_EL0	Performance Monitors Overflow Flag Status Set Register
PMPCSR	Program Counter Sample Register
PMPIDR0	Performance Monitors Peripheral Identification Register 0
PMPIDR1	Performance Monitors Peripheral Identification Register 1
PMPIDR2	Performance Monitors Peripheral Identification Register 2
PMPIDR3	Performance Monitors Peripheral Identification Register 3
PMPIDR4	Performance Monitors Peripheral Identification Register 4
PMSWINC_EL0	Performance Monitors Software Increment Register
PMVIDSR	VMID Sample Register

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H4.5 Functional index of memory-mapped registers

This section is an index of the memory-mapped registers, divided by functional group. Each of the following sections lists the registers for a functional group:

- ID registers.
- Performance monitors registers.
- Generic timer registers.
- Debug registers.
- Cross-trigger interface registers.

H4.5.1 ID registers

This section is an index to the registers in the Identification registers functional group.

Table H4-17 ID registers

Register	Description, see
EDAA32PFR	External Debug Auxiliary Processor Feature Register
EDDFR	External Debug Feature Register
EDPFR	External Debug Processor Feature Register
MIDR_EL1	Main ID Register

H4.5.2 Performance monitors registers

This section is an index to the registers in the Performance Monitors registers functional group.

Table H4-18 Performance monitors registers

Register	Description, see
PMAUTHSTATUS	Performance Monitors Authentication Status register
PMCCFILTR_EL0	Performance Monitors Cycle Counter Filter Register
PMCCNTR_EL0	Performance Monitors Cycle Counter
PMCEID0	Performance Monitors Common Event Identification register 0
PMCEID1	Performance Monitors Common Event Identification register 1
PMCEID2	Performance Monitors Common Event Identification register 2
PMCEID3	Performance Monitors Common Event Identification register 3
PMCFGR	Performance Monitors Configuration Register
PMCID1SR	CONTEXTIDR_EL1 Sample Register
PMCID2SR	CONTEXTIDR_EL2 Sample Register
PMCIDR0	Performance Monitors Component Identification Register 0

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
PMCIDR1	Performance Monitors Component Identification Register 1
PMCIDR2	Performance Monitors Component Identification Register 2
PMCIDR3	Performance Monitors Component Identification Register 3
PMCNTENCLR_EL0	Performance Monitors Count Enable Clear Register
PMCNTENSET_EL0	Performance Monitors Count Enable Set Register
PMCR_EL0	Performance Monitors Control Register
PMDEVAFF0	Performance Monitors Device Affinity register 0
PMDEVAFF1	Performance Monitors Device Affinity register 1
PMDEVARCH	Performance Monitors Device Architecture register
PMDEVID	Performance Monitors Device ID register
PMDEVTYPE	Performance Monitors Device Type register
PMIIDR	Performance Monitors Implementation Identification Register
PMINTENCLR_EL1	Performance Monitors Interrupt Enable Clear Register
PMINTENSET_EL1	Performance Monitors Interrupt Enable Set Register
PMITCTRL	Performance Monitors Integration mode Control register
PMLAR	Performance Monitors Lock Access Register
PMLSR	Performance Monitors Lock Status Register
PMMIR	Performance Monitors Machine Identification Register
PMOVSCLR_EL0	Performance Monitors Overflow Flag Status Clear register
PMOVSSET_EL0	Performance Monitors Overflow Flag Status Set Register
PMPCSR	Program Counter Sample Register
PMPIDR0	Performance Monitors Peripheral Identification Register 0
PMPIDR1	Performance Monitors Peripheral Identification Register 1
PMPIDR2	Performance Monitors Peripheral Identification Register 2
PMPIDR3	Performance Monitors Peripheral Identification Register 3
PMPIDR4	Performance Monitors Peripheral Identification Register 4
PMSWINC_EL0	Performance Monitors Software Increment Register
PMVIDSR	VMID Sample Register

H4.5.3 Debug registers

This section is an index to the registers in the Debug registers functional group.

Table H4-19 Debug registers

Register	Description, see
DBGAUTHSTATUS_EL1	Debug Authentication Status Register

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
DBGCLAIMCLR_EL1	Debug CLAIM Tag Clear Register
DBGCLAIMSET_EL1	Debug CLAIM Tag Set Register
DBGDTRRX_EL0	Debug Data Transfer Register, Receive
DBGDTRTX_EL0	Debug Data Transfer Register, Transmit
EDACR	External Debug Auxiliary Control Register
EDCIDR0	External Debug Component Identification Register 0
EDCIDR1	External Debug Component Identification Register 1
EDCIDR2	External Debug Component Identification Register 2
EDCIDR3	External Debug Component Identification Register 3
EDDEVAFF0	External Debug Device Affinity register 0
EDDEVAFF1	External Debug Device Affinity register 1
EDDEVARCH	External Debug Device Architecture Register
EDDEVID	External Debug Device ID register 0
EDDEVID1	External Debug Device ID Register 1
EDDEVID2	External Debug Device ID register 2
EDDEVTYPE	External Debug Device Type register
EDECCR	External Debug Exception Catch Control Register
EDECR	External Debug Execution Control Register
EDESR	External Debug Event Status Register
EDITCTRL	External Debug Integration mode Control register
EDITR	External Debug Instruction Transfer Register
EDLAR	External Debug Lock Access Register
EDLSR	External Debug Lock Status Register
EDPIDR0	External Debug Peripheral Identification Register 0
EDPIDR1	External Debug Peripheral Identification Register 1
EDPIDR2	External Debug Peripheral Identification Register 2
EDPIDR3	External Debug Peripheral Identification Register 3
EDPIDR4	External Debug Peripheral Identification Register 4
EDPRCR	External Debug Power-Reset Control Register
EDPRSR	External Debug Processor Status Register
EDRCR	External Debug Reserve Control Register
EDSCR	External Debug Status and Control Register
EDWAR	External Debug Watchpoint Address Register

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

OSLAR_EL1	OS Lock Access Register

H4.5.4 Cross-trigger interface registers

This section is an index to the registers in the Cross-Trigger Interface registers functional group.

Table H4-20 Cross-trigger interface registers

Register	Description, see
ASICCTL	CTI External Multiplexer Control register
CTIAPPCLEAR	CTI Application Trigger Clear register
CTIAPPPULSE	CTI Application Pulse register
CTIAPPSET	CTI Application Trigger Set register
CTIAUTHSTATUS	CTI Authentication Status register
CTICHINSTATUS	CTI Channel In Status register
CTICHOUTSTATUS	CTI Channel Out Status register
CTICIDR0	CTI Component Identification Register 0
CTICIDR1	CTI Component Identification Register 1
CTICIDR2	CTI Component Identification Register 2
CTICIDR3	CTI Component Identification Register 3
CTICLAIMCLR	CTI CLAIM Tag Clear register
CTICLAIMSET	CTI CLAIM Tag Set register
CTICONTROL	CTI Control register
CTIDEVAFF0	CTI Device Affinity register 0
CTIDEVAFF1	CTI Device Affinity register 1
CTIDEVARCH	CTI Device Architecture register
CTIDEVCTL	CTI Device Control register
CTIDEVID	CTI Device ID register 0
CTIDEVID1	CTI Device ID register 1
CTIDEVID2	CTI Device ID register 2
CTIDEVTYPE	CTI Device Type register
CTIGATE	CTI Channel Gate Enable register
CTIINTACK	CTI Output Trigger Acknowledge register
CTIITCTRL	CTI Integration mode Control register
CTILAR	CTI Lock Access Register
CTILSR	CTI Lock Status Register
CTIPIDR0	CTI Peripheral Identification Register 0
CTIPIDR1	CTI Peripheral Identification Register 1
CTIPIDR2	CTI Peripheral Identification Register 2
CTIPIDR3	CTI Peripheral Identification Register 3
CTIPIDR4	CTI Peripheral Identification Register 4

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
CTITRIGINSTATUS	CTI Trigger In Status register
CTITRIGOUTSTATUS	CTI Trigger Out Status register

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Glossary

A64 instruction A word that specifies an operation to be performed by a PE that is executing in an Exception level that is using AArch64. A64 instructions must be word-aligned.

AArch32 The 32-bit Execution state. AArch32 state and the associated instrucion sets, T32 and A32, are not described in this Manual. See *Arm Architecture Reference Manual, for A-profile architecture (ARM DDI0487)*.

AArch64 The 64-bit Execution state. In AArch64 state, addresses are held in 64-bit registers, and instructions in the base instruction set can use 64-bit registers for their processing. AArch64 state supports the A64 instruction set.

See also A64 instruction.

Abort An exception caused by an illegal memory access. Aborts can be caused by the external memory system or the MMU.

Active element An Active element is an SVE vector element or predicate element that is a source register element or destination register element used by an SVE instruction. When the corresponding element in the instruction's Governing predicate is TRUE, the element is Active. If an instruction is unpredicated, all of the vector elements or predicate elements are implicitly Active.

Addressing mode Means a method for generating the memory address used by a load/store instruction.

Advanced SIMD A feature of the Arm architecture that provides SIMD operations on a register file of SIMD and floating-point registers. Where an implementation supports both Advanced SIMD and floating-point instructions, these instructions operate on the same register file.

Aligned A data item stored at an address that is exactly divisible by the highest power of 2 that divides exactly into its size in bytes. Aligned halfwords, words and doublewords therefore have addresses that are divisible by 2, 4 and 8 respectively.

An aligned access is one where the address of the access is aligned to the size of each element of the access.

AMBA Advanced Microcontroller Bus Architecture. The AMBA family of protocol specifications is the Arm open standard for on-chip buses. AMBA provides solutions for the interconnection and management of the functional blocks that make up a *System-on-Chip* (SoC). Applications include the development of embedded systems with one or more processors or signal processors and multiple peripherals.

Analysis of the trace element stream Refers to the process of:

- Tracing elements that carry information that a trace analyzer requires to enable it to analyze the trace successfully.
- Tracing elements that either directly indicate program execution, or carry information about program execution.

A trace element stream might also contain trace elements that contain timing information. This term is distinct from analysis of program execution.

Architecturally executed An instruction is architecturally executed only if it would be executed in a simple sequential execution of the program. When such an instruction has been executed and retired, it has been *architecturally executed*. Any instruction that, in a simple sequential execution of a program, is treated as a `NOP` because it fails its condition code check, is an architecturally executed instruction.

In a PE that performs speculative execution, an instruction is not architecturally executed if the PE discards the results of a speculative execution.

See also Condition code check, Simple sequential execution.

Architecturally mapped Where this manual describes a register as being *architecturally mapped* to another register, this indicates that, in an implementation that supports both of the registers, the two registers access the same state.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Architecturally UNKNOWN An architecturally UNKNOWN value is a value that is not defined by the architecture but must meet the requirements of the definition of UNKNOWN. Implementations can define the value of the field, but are not required to do so.

See also IMPLEMENTATION DEFINED.

Architecturally Allowed Execution An *Architecturally Allowed Execution* must satisfy the External visibility requirement.

Associativity See Cache associativity.

Asynchronous accumulation Faults that are accumulated in a status register, where the update to the register is asynchronous to the instruction that causes the fault.

Atomicity Describes either single-copy atomicity or multi-copy atomicity. Atomicity in the Arm architecture defines these forms of atomicity for the Arm architecture.

See also Multi-copy atomicity and Single-copy atomicity.

Might also refer to poison-atomicity. See Poison atomicity.

Availability Readiness for correct service.

Banked register A register that has multiple instances, with the instance that is in use depending on the PE mode, Security state, or other PE state.

Baseboard Management Controller A PE dedicated to system control and monitoring.

Base register A register specified by a load/store instruction that is used as the base value for the address calculation for the instruction. Depending on the instruction and its addressing mode, an offset can be added to or subtracted from the base register value to form the virtual address that is sent to memory.

Base register writeback Describes writing back a modified value to the base register used in an address calculation.

Behaves as if Where this manual indicates that a PE *behaves as if* a certain condition applies, all descriptions of the operation of the PE must be re-evaluated taking account of that condition, together with any other conditions that affect operation.

Big-endian memory Means that, for example:

- A byte or halfword at a word-aligned address is the most significant byte or halfword in the word at that address.
- A byte at a halfword-aligned address is the most significant byte in the halfword at that address.

See also Endianness, Little-endian memory.

Blocking Describes an operation that does not permit following instructions to be executed before the operation completes.

A non-blocking operation can permit following instructions to be executed before the operation completes, and in the event of encountering an exception does not signal an exception to the PE. This enables implementations to retire following instructions while the non-blocking operation is executing, without the need to retain precise PE state.

Branch History Buffer (BHB) The Branch History Buffer is an internal structure that stores branch instruction information including the history of previously executed branches.

Branch prediction Is where a PE selects a future execution path to fetch along. For example, after a branch instruction, the PE can choose to speculatively fetch either the instruction following the branch or the instruction at the branch target.

See also Prefetching.

Breakpoint A debug event triggered by the execution of a particular instruction, specified by one or both of the address of the instruction and the state of the PE when the instruction is executed.

Built-in self test (BIST) A mechanism that permits a machine to test itself.

Burst A group of transfers that form a single transaction.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

BWA BandWidth Allocation.

BWPBM BandWidth Portion Bit Map.

Byte An 8-bit data item.

CE See Corrected Error.

Cache associativity The number of locations in a cache set to which an address can be assigned. Each location is identified by its *way* value.

Cache level The position of a cache in the cache hierarchy. In the Arm architecture, the lower numbered levels are those closest to the PE. For more information, see Terms used in describing the cache maintenance instructions.

Cache line The basic unit of storage in a cache. Its size in words is always a power of two, usually 4 or 8 words. A cache line must be aligned to a suitable memory boundary. A *memory cache line* is a block of memory locations with the same size and alignment as a cache line. Memory cache lines are sometimes loosely called cache lines.

Cache lockdown Enables critical software and data to be loaded into the cache so that the cache lines containing them are not subsequently reallocated. It alleviates the delays caused by accessing a cache in a worst-case situation. This ensures that all subsequent accesses to the software and data concerned are cache hits and so complete quickly.

Cache miss A memory access that cannot be processed at high speed because the data it addresses is not in the cache.

Cache sets Areas of a cache, divided up to simplify and speed up the process of determining whether a cache hit occurs. The number of cache sets is always a power of two.

Cache way A cache way consists of one cache line from each cache set. The cache ways are indexed from 0 to (Associativity-1). Each cache line in a cache way is chosen to have the same index as the cache way. For example, cache way *n* consists of the cache line with index *n* from each cache set.

Coherence order See Coherent.

Coherent Data accesses from a set of observers to a byte in memory are coherent if accesses to that byte in memory by the members of that set of observers are consistent with there being a single total order of all writes to that byte in memory by all members of the set of observers. This single total order of all to writes to that memory location is the *coherence order* for that byte in memory.

Commit window The Commit window defines the range of *P0 elements* that are committed by a *Commit element*. The oldest *P0 element* in the Commit window is the first *P0 element* committed when a *Commit element* occurs. By default, the Commit window starts on the oldest uncommitted *P0 element* and moves forward to the next uncommitted *P0 element*, with each *P0 element* committed by a *Commit element*. The Commit Window Move element moves the start of the Commit window by a number of *P0 elements*, to allow a *Commit element* to commit *P0 elements* that are younger than the oldest uncommitted *P0 elements*, leaving these older *P0 elements* uncommitted.

Completer An agent in a computing system that responds to and completes a memory transaction that was initiated by a Requester.

See also Requester.

Constructive instruction encoding A constructive instruction encoding is an instruction encoding where the destination register is encoded independently of the source registers.

Condition code check The process of determining whether a conditional instruction executes normally or is treated as a `NOP`. For an instruction that includes a condition code field, that field is compared with the condition flags to determine whether the instruction is executed normally.

See also Condition code field, Condition flags, Conditional execution.

Condition code field A 4-bit field in an instruction that specifies the condition under which the instruction executes.

See also Condition code check.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Condition flags The N, Z, C, and V bits of PSTATE, an SPSR, or FPSCR. See the register descriptions for more information.

See also Condition code check, PSTATE.

Conditional execution When a conditional instruction starts executing, if the condition code check returns TRUE, the instruction executes normally. Otherwise, it is treated as a `NOP`.

See also Condition code check.

CONSTRAINED UNPREDICTABLE Where an instruction can result in UNPREDICTABLE behavior, the Armv8 and later architectures specify a narrow range of permitted behaviors. This range is the range of CONSTRAINED UNPREDICTABLE behavior. All implementations that are compliant with the architecture must follow the CONSTRAINED UNPREDICTABLE behavior.

Execution at Non-secure EL1 or EL0 of an instruction that is CONSTRAINED UNPREDICTABLE can be implemented as generating a trap exception that is taken to EL2, provided that at least one instruction that is not UNPREDICTABLE and is not CONSTRAINED UNPREDICTABLE causes a trap exception that is taken to EL2.

In body text, the term CONSTRAINED UNPREDICTABLE is shown in SMALL CAPITALS.

See also UNPREDICTABLE.

Containable error See Contained error.

Contained error An error that is not uncontained or uncontainable.

Context switch The saving and restoring of computational state when switching between different threads or processes. In this manual, the term context switch describes any situation where the context is switched by an operating system and might or might not include changes to the address space.

Context Synchronization event One of:

- Performing an ISB operation. An ISB operation is performed when an `ISB` instruction is executed and does not fail its condition code check.
- Exception entry, if any of the following is true:
  - FEAT_ExS is not implemented.
  - FEAT_ExS is implemented and the appropriate SCTLR_ELx.EIS bit is set.
- Return from an exception, if any of the following is true:
  - FEAT_ExS is not implemented.
  - FEAT_ExS is implemented and the appropriate SCTLR_ELx.EIS bit is set.
  - Exit from Debug state.
- Executing a `DCPS` instruction.
- Executing a `DRPS` instruction.

The effects of a Context synchronization event are:

- All unmasked interrupts that are pending at the time of the Context synchronization event are taken before the first instruction after the Context synchronization event.
- If halting is allowed, all Halting debug events that are pending at the time of the Context synchronization event are taken before the first instruction after the Context synchronization event.
- No instructions appearing in program order after an instruction that causes a Context synchronization event will have performed any part of their functionality until the Context synchronization event has occurred.
- All direct and indirect writes to System registers that are made before the Context synchronization event affect any instruction, including a direct read, that appears in program order after the instruction causing the Context synchronization event.
- All completed changes to the translation tables for entries that, before the change, were not permitted to be cached in a TLB, affect all instruction fetches that appear in program order after the instruction causing the Context synchronization event.
- The following invalidations, if completed before the Context synchronization event will affect all instructions that appear in program order after the Context synchronization event:

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - TLBS.
  - Instruction caches.
- Changes to the Debug external authentication interfaces that are made before the Context synchronization event affect any instruction that appears in program order after the instruction causing the Context synchronization event.
- The effect of the completion of any of the instructions added by FEAT_SPECRES is synchronized to the current execution context.
- Restrictions on the effects of speculation (as described in Restrictions on the effects of speculation) are observed.
- Ensuring that the `TSB` instruction is executed in the necessary order with respect to other instructions.
- Profiling operations for all instructions that are executed in program order are synchronized by execution of a `PSB` instruction before the Context synchronization event.
- See also:
  - Restrictions on the effects of speculation.
  - Trace Synchronization Barrier.
  - Synchronization and debug exceptions.
  - Synchronization in self-hosted trace.
  - Synchronization of register updates.
  - Speculation in the trace element stream.
  - Trace Synchronization event.
  - Self-hosted trace extension synchronization rules.
  - Execution, data prediction and prefetching restriction System instructions.
  - Synchronization and Statistical Profiling.
  - Synchronization and Halting debug events.
  - Synchronization of memory-mapped registers.
  - CONSTRAINED UNPREDICTABLE behavior due to inadequate context synchronization.

Note

- The architecture requires that instructions that generate Context synchronization events do not appear to be executed speculatively, except that the performance monitor counters are permitted to reveal such speculation.
- *Context synchronization events* were previously described as *context synchronization operations*.

Conventional memory Memory locations from which generic OSs and application runtimes expect to create allocations for general software use.

Core See Processing element (PE).

CoreSight Address Translation Unit A form of System MMU for trace streams.

Corrected error An error that is detected by hardware and that hardware has corrected.

Counter resolution See Counter resolution.

CPBM Cache-Portion Bit Map.

CSU Cache-Storage Usage.

Data independent timing (DIT) The time that it takes to execute a piece of code where the time is not a function of the data being operated on. For more information, see About PSTATE.DIT and About the DIT bit.

Debugger In most of this manual, *debugger* refers to any agent that is performing debug. However, some chapters or parts of this manual require a more rigorous definition, and define debugger locally. See:

- Definition of a debugger in the context of self-hosted debug.
- Definition of a debugger in the context of self-hosted debug.
- Definition and constraints of a debugger in the context of external debug.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Deferred error An error that has not been silently propagated but does not require immediate action at the producer. The error might have passed from the producer to a consumer.

Deprecated Something that is present in the Arm architecture for backward compatibility.

Features that are deprecated but are not OPTIONAL are present in current implementations of the Arm architecture, but might not be present, or might be deprecated and OPTIONAL, in future versions of the Arm architecture.

Arm strongly recommends software to avoid using deprecated features. Where a feature is deprecated for performance reasons, Arm strongly recommends software to avoid using that feature because of the likelihood of the significant negative impact on performance.

See also OPTIONAL.

Destructive instruction encoding A destructive instruction encoding is an instruction encoding where one of the source registers is also used as the destination register.

Detected error An error that has been detected and signaled to a consumer.

Detected Uncorrected error A detected error that has not been be corrected and causes failure.

Digital signal processing (DSP) Algorithms for processing signals that have been sampled and converted to digital form. DSP algorithms often use saturated arithmetic.

Direct branch Branch instructions whose target is relative to the PC of the branch and the offset is determined solely from the opcode of the executed instruction.

See also Indirect branch.

Direct Memory Access (DMA) An operation that accesses main memory directly, without the PE performing any accesses to the data concerned.

Direct read A direct read of a System register is a read performed by a System register access instruction.

For more information, see Direct read.

See also Direct write, Indirect read, Indirect write.

Direct write A direct write of a System register is a write performed by a System register access instruction.

For more information, see Direct write.

See also Direct read, Indirect read, Indirect write.

DMA See Direct Memory Access (DMA).

DNM See Do-Not-Modify (DNM).

Domain In the Arm architecture, *domain* is used in the following contexts.

Shareability domain Defines a set of observers for which the Shareability attributes make the data or unified caches transparent for data accesses.

Power domain Defines a block of logic with a single, common, power supply.

Memory regions domain

When using the Short-descriptor translation table format, defines a collection of Sections, Large pages and Small pages of memory, that can have their access permissions switched rapidly by writing to the *Domain Access Control Register* (DACR). Arm deprecates any use of memory regions domains.

Do-Not-Modify (DNM) Means the value must not be altered by software. DNM fields read as UNKNOWN values, and must only be written with the value read from the same field on the same PE.

Double-precision value Consists of two consecutive 32-bit words that are interpreted as a basic double-precision floating-point number according to the *IEEE Standard for Floating-point Arithmetic*.

Doubleword A 64-bit data item. Doublewords are normally at least word-aligned in Arm systems.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Doubleword-aligned Means that the address is divisible by 8.

Downstream Information propagating in the direction from Requesters towards terminating Completer components.

DSB Data Synchronization Barrier.

DSP See Digital signal processing (DSP).

DUE See Detected Uncorrected error.

DUE FIT RATE The FIT rate for failures from a DUE.

ECC See Error Correction Code (ECC).

EDAC See Error Correction Code (ECC).

EDC See Error Detection Code (EDC).

Effective frequency See Effective frequency.

Effective value A register control field, meaning a field in a register that controls some aspect of the behavior, can be described as having an *Effective value*:

- In some cases, the description of a control *a* specifies that when control *a* is active it causes a register control field *b* to be treated as having a fixed value for all purposes other than direct reads, or direct reads and direct writes, of the register containing control field *b*. When control *a* is active that fixed value is described as the *Effective value* of register control field *b*.
  In other cases, in some contexts a register control field *b* is not implemented or is not accessible, but behavior of the PE is as if control field *b* was implemented and accessible, and had a particular value. In this case, that value is the *Effective value* of register control field *b*.

  Note

  Where a register control field is introduced in a particular version of the architecture, and is not implemented in an earlier version of the architecture, typically it will have an *Effective value* in that earlier version of the architecture.

- Otherwise, the *Effective value* of a register control field is the value of that field.

Endianness An aspect of the system memory mapping.

See also Big-endian memory and Little-endian memory.

Error Correction Code (ECC) A code capable of detecting and correcting a number of errors.

Error Detection Code (EDC) A code capable of detecting, but not correcting, errors.

Error propagation Passing an error from a producer to a consumer.

Error record Data recorded about an error, usually by hardware.

ETEEvent A feature of the trace unit that is used to generate *Event elements* and drive External Outputs. Each ETEEvent can be programmed to be sensitive to resource events.

Event trace The trace uses *Event elements* that indicate certain events have occurred in the program that the PE is executing. The program events to be indicated are selected before a trace session.

Exceptional occurrence Events indicated by an *Exception element* by the ETE architecture, including the following:

- PE architectural exceptions.
- ETE defined exceptions.
- IMPLEMENTATION DEFINED exceptions.

Exception Handles an event. For example, an exception could handle an external interrupt or an undefined instruction.

Exception vector A fixed address that contains the address of the first instruction of the corresponding exception handler.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Executing processing element (PEe) The name given to an example PE that executes a specified instruction.

See also Processing element (PE).

Execution stream The stream of instructions that would have been executed by sequential execution of the program.

External abort An abort that is generated by the external memory system.

Fault An event that causes the program to deviate from the intended behavior, for example an invalid translation.

Field Replaceable Unit (FRU) A component or unit in a system that can be replaced without return to base.

First active element The First active element of an SVE vector or predicate register is the lowest numbered element that is an Active element.

First-fault load SVE provides a First-fault option for some SVE vector load instructions. This option causes memory access faults to be suppressed if they do not occur as a result of the First active element of the vector. Instead, the FFR is updated to indicate which of the Active elements were not successfully loaded.

Flat address mapping Is where the physical address for every access is equal to its virtual address.

Flush-to-zero mode A processing mode that optimizes the performance of some floating-point algorithms by replacing the denormalized operands and Intermediate results with zeros, without significantly affecting the accuracy of their final results.

FRU See Field Replaceable Unit (FRU).

General-purpose registers The registers that the base instructions use for processing:

- In AArch64 state the general-purpose registers are:
  - W0-W30 when accessed as 32-bit registers.
  - X0-X30 when accessed as 64-bit registers.

Generated by The memory model is written in terms of reads from memory and writes to memory. These reads and writes:

- Are generated by instructions such as loads, stores, and atomic memory accesses.
- Correspond to the memory accesses, other than translation table walks, that are defined in the instruction pseudocode.

Some instructions generate more than one read or write.

Generic Interrupt Controller (GIC) Arm system architecture interrupt controller for IRQ and FIQ interrupt exceptions.

Governing predicate The predicate register that is used to determine the Active elements of a predicated instruction is known as the Governing predicate for that instruction.

GIC See Generic Interrupt Controller (GIC).

Halfword A 16-bit data item. Halfwords are normally halfword-aligned in Arm systems.

Halfword-aligned Means that the address is divisible by 2.

High vectors An alternative location for the exception vectors. The high vector address range is near the top of the address space, rather than at the bottom.

ICN InterConnect Network.

IGNORED Indicates that the architecture guarantees that the bit or field is not interpreted or modified by hardware.

In body text, the term IGNORED is shown in SMALL CAPITALS.

Immediate and offset fields Are unsigned unless otherwise stated.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Immediate value A value that is encoded directly in the instruction and used as numeric data when the instruction is executed. Many instructions can be used with an immediate argument.

IMP An abbreviation used in diagrams to indicate that one or more bits have IMPLEMENTATION DEFINED behavior.

IMPLEMENTATION DEFINED Means that the behavior is not architecturally defined, but must be defined and documented by individual implementations.

In body text, the term IMPLEMENTATION DEFINED is shown in SMALL CAPITALS.

IMPLEMENTATION SPECIFIC Means that the behavior is not architecturally defined, and might not be documented by an individual implementation. Used when there are several implementation options available and the option chosen does not affect software compatibility.

In body text, the term IMPLEMENTATION SPECIFIC is shown in SMALL CAPITALS.

Imprecise exception An exception that is not precise.

Inactive element An Inactive element is an SVE vector element or predicate element that is an unused source register element or destination register element for the associated instruction. When the corresponding element of an instruction's Governing predicate is FALSE, it is an Inactive element.

Index register A register specified in some load and store instructions. The value of this register is used as an offset to be added to or subtracted from the base register value to form the virtual address that is sent to memory. Some instruction forms permit the index register value to be shifted before the addition or subtraction.

Indirect branch Branch instructions that are not direct branches.

See also Direct branch.

Indirect read When an instruction uses a System register value to establish operating conditions, that use of the System register is an indirect read of the System register.

For more information, including additional examples of indirect reads, see Indirect read.

See also Direct read, Direct write, Indirect write.

Indirect write An indirect write of a System register occurs when the contents of a register are updated by some mechanism other than a Direct write to that register. For example, an indirect write to a register might occur as a side-effect of executing an instruction that does not perform a direct write to the register, or because of some operation performed by an external agent.

For more information, see Indirect write.

See also Direct read, Direct write, Indirect read.

Inline literals These are constant addresses and other data items held in the same area as the software itself. They are automatically generated by compilers, and can also appear in assembler code.

Intermediate physical address (IPA) An implementation of virtualization, the address to which a Guest OS maps a VA. A hypervisor might then map the IPA to a PA. Typically, the Guest OS is unaware of the translation from IPA to PA.

See also Physical address (PA), Virtual address (VA).

Interrupt In a PE context, an asynchronous exception. There are three interrupt exceptions: IRQ, FIQ, and SError. IRQ and FIQ are always precise. In a system architecture context, an asynchronous event sent to a PE or GIC for processing as an interrupt exception.

IPA See Intermediate physical address (IPA).

kvm Kernel-based Virtual Machine, an open-source software package that implements a type-2 hypervisor within Linux.

Latent error An error that is present in a system but not yet detected.

Last active element The Last active element of an SVE vector or predicate register is the highest numbered element that is an Active element.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Level See Cache level.

Level of Coherence (LoC) The last level of cache that must be cleaned or invalidated when cleaning or invalidating to the point of coherency. For more information, see Terms used in describing the cache maintenance instructions.

See also Cache level, Point of coherency (PoC).

Level of Unification, Inner Shareable (LoUIS) The last level of cache that must be cleaned or invalidated when cleaning or invalidating to the point of unification for the Inner Shareable Shareability domain. For more information, see Terms used in describing the cache maintenance instructions.

See also Cache level, Point of unification (PoU).

Level of Unification, uniprocessor (LoUU) For a PE, the last level of cache that must be cleaned or invalidated when cleaning or invalidating to the point of unification for that PE. For more information, see Terms used in describing the cache maintenance instructions.

See also Cache level, Point of unification (PoU).

Line See Cache line.

Little-endian memory Means that, for example:

- A byte or halfword at a word-aligned address is the least significant byte or halfword in the word at that address.
- A byte at a halfword-aligned address is the least significant byte in the halfword at that address.

See also Big-endian memory, Endianness.

Load/store architecture An architecture where data-processing operations only operate on register contents, not directly on memory contents.

LoC See Level of Coherence (LoC).

LoUIS See Level of Unification, Inner Shareable (LoUIS).

LoUU See Level of Unification, uniprocessor (LoUU).

Lockdown See Cache lockdown.

MBWU Memory BandWidth Usage.

Memory barrier See Memory barriers.

Memory coherency The problem of ensuring that when a memory location is read, either by a data read or an instruction fetch, the value actually obtained is always the value that was most recently written to the location. This can be difficult when there are multiple possible physical locations, such as main memory and at least one of a write buffer and one or more levels of cache.

Memory element An item of data in memory that is transferred to or from a vector element by an SVE load or store instruction. Each memory element has an access size and a type. The memory element access size is specified by each load and store instruction independently of the vector element size.

Memory Management Unit (MMU) Provides detailed control of the part of a memory system that provides a single stage of address translation. Most of the control is provided using translation tables that are held in memory, and define the attributes of different regions of the physical memory map.

Memory Protection Unit (MPU) A hardware unit whose registers provide simple control of a limited number of protection regions in memory.

Merging predication When a predicated SVE instruction specifies merging predication, the Inactive elements of the destination register remain unchanged.

Memory system component MSC. A function, unit, or design block in a memory system that can have partitionable resources. MSCs consist of all units that handle load or store requests issued by any MPAM Requester. These can include cache memories, interconnects, memory management units, memory channel

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

controllers, queues, buffers, and rate adaptors. An MSC can contain one or more resources that each can have zero or more resource partitioning controls. For example, a PE can contain several caches, each of which might have zero or more resource partitioning controls.

Memory system resource A resource that affects the performance of software's use of the memory system and is either local to an MSC (such as cache-memory capacity) or non-local (such as memory bandwidth, which is present over an entire path, from Requester to Completer, that can pass through multiple MSCs).

MMR Memory-mapped Register.

Microarchitecturally-finished An operation that has finished all of its operational pseudocode, although the results of any memory accesses, including translation table walks and updates, are not yet coherent with other observers.

Microarchitecturally-unfinished An operation that has not completed all of its operational pseudocode.

Miss See Cache miss.

MMU See Memory Management Unit (MMU).

MPU See Memory Protection Unit (MPU).

Multi-copy atomicity The form of atomicity described in Requirements for multi-copy atomicity.

See also Atomicity, Single-copy atomicity.

NaN Not a Number. A floating-point value that can be used when neither a numeric value nor an infinity is appropriate. A NaN can be a *Quiet* NaN, that propagate through most floating-point operations, or a *signaling* NaN, that causes an Invalid Operation floating-point exception when used. For more information, see the *IEEE Standard for Floating-point Arithmetic*.

See also Quiet NaN, Signaling NaN.

Natural eviction A natural eviction is an eviction that occurs in the course of the normal operation of the memory system, rather than because of an operation that explicitly causes an eviction from the cache, such as the execution of a cache maintenance instruction. Typically, a natural eviction occurs when the caching algorithm requires data to be cached but the cache does not have room for that data.

Non-fault load SVE provides a Non-fault option for some SVE vector load instructions. This option causes all memory access faults to be suppressed. Instead, the FFR is updated to indicate which of the Active elements were not successfully loaded.

Observer A PE or mechanism in the system, such as a peripheral device, that can generate reads from or writes to memory.

Obsolete Obsolete indicates something that is no longer supported by Arm. When an architectural feature is described as obsolete, this indicates that the architecture has no support for that feature, although an earlier version of the architecture did support it.

Offset addressing Means that the memory address is formed by adding or subtracting an offset to or from the base register value.

OPTIONAL When applied to a feature of the architecture, OPTIONAL indicates a feature that is not required in an implementation of the Arm architecture:

- If a feature is OPTIONAL and deprecated, this indicates that the feature is being phased out of the architecture. Arm expects such a features to be included in a new implementation only if there is a known backward-compatibility reason for the inclusion of the feature.
  A feature that is OPTIONAL and deprecated might not be present in future versions of the architecture.
- A feature that is OPTIONAL but not deprecated is, typically, a feature added to a version of the Arm architecture after the initial release of that version of the architecture. Arm recommends that such features are included in all new implementations of the architecture.

In body text, these meanings of the term OPTIONAL are shown in SMALL CAPITALS.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Do not confuse these Arm-specific uses of OPTIONAL with other uses of *optional*, where it has its usual meaning. These include:

- Optional arguments in the syntax of many instructions.
- Behavior determined by an implementation choice

See also Deprecated.

Other-multi-copy atomic This is a Memory Write effect from an observer that, if observed by a different observer, is then observed by all other observers that access the location coherently.

PA See Physical address (PA).

Packed access A memory access that is performed as a result of an SVE load or store instruction for which the vector element size and the memory element size are the same.

PCIe See Peripheral Component Interconnect Express (PCI Express or PCIe).

PE See Processing element (PE).

Peripheral Component Interconnect Express (PCI Express or PCIe) A high-speed serial computer expansion bus standard maintained and developed by the PCI Special Interest Group.

PEe The executing processing element. See Executing processing element.

PFA See Predictive Failure Analysis (PFA).

Physical address (PA) An address that identifies a location in the physical memory map.

See also Intermediate physical address (IPA), Virtual address (VA).

Point of coherency (PoC) See Terms used in describing the cache maintenance instructions for the AArch64 state and Terminology for Clean, Invalidate, and Clean and Invalidate instructions for the AArch32 state.

Point of Deep Persistence (PoDP) See Terms used in describing the cache maintenance instructions.

Point of encryption (PoE) See Terms used in describing the cache maintenance instructions.

Point of persistence (PoP) See Terms used in describing the cache maintenance instructions.

Point of physical aliasing (PoPA) See Terms used in describing the cache maintenance instructions.

Point of physical storage (PoPS) See Terms used in describing the cache maintenance instructions.

Point of unification (PoU) See Terms used in describing the cache maintenance instructions for the AArch64 state and Terminology for Clean, Invalidate, and Clean and Invalidate instructions for the AArch32 state.

Poison atomicity The form of atomicity described in Clearing Deferred errors from poisoned Locations.

See also Atomicity.

Poison granule A region of naturally-aligned memory affected by poison.

See Error propagation for information on poison.

Post-indexed addressing Means that the memory address is the base register value, but an offset is added to or subtracted from the base register value and the result is written back to the base register.

Precise exception An exception where the exception handler receives the state of the PE and the state of the memory system consistent with the PE having executed all of the instructions up to, but not including, the point in the instruction stream where the exception was taken. The state of the PE and the state of the memory do not include instructions that occurred after this point.

Prefetching Prefetching refers to speculatively fetching instructions or data from the memory system. In particular, instruction prefetching is the process of fetching instructions from memory before the instructions that

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

precede them, in simple sequential execution of the program, have finished executing. Prefetching an instruction does not mean that the instruction has to be executed.

In this manual, references to instruction or data fetching apply also to prefetching, unless the context explicitly indicates otherwise.

Note

The Prefetch Abort exception can be generated on any instruction fetch, and is not limited to speculative instruction fetches.

See also Simple sequential execution.

Prefixed instruction The instruction that immediately follows a MOVPRFX instruction in program order.

Pre-indexed addressing Means that the memory address is formed in the same way as for offset addressing, but the memory address is also written back to the base register.

Processing element (PE) The abstract machine defined in the Arm architecture, as documented in an Arm Architecture Reference Manual. A PE implementation compliant with the Arm architecture must conform with the behaviors described in the corresponding Arm Architecture Reference Manual.

Protection granule A quantum of memory for which an EDC or ECC provides detection or correction. For example, a 72/64 SECDED ECC scheme has a 64-bit protection granule.

Protection region A memory region whose position, size, and other properties are defined by Memory Protection Unit registers.

Protection Unit See Memory Protection Unit (MPU).

Pseudo-instruction UAL assembler syntax that assembles to an instruction encoding that is expected to disassemble to a different assembler syntax, and is described in this manual under that other syntax. For example, `MOV <Rd>, <Rm>, LSL #<n>` is a pseudo-instruction that is expected to disassemble as `LSL <Rd>, <Rm>, #<n>`.

PSTATE An abstraction of process state information. All of the instruction sets provide instructions that operate on elements of PSTATE.

See also Condition flags.

Quadword A 128-bit data item. Quadwords are normally at least word-aligned in Arm systems.

Quadword-aligned Means that the address is divisible by 16.

Quiet NaN A NaN that propagates unchanged through most floating-point operations.

See also NaN, Signaling NaN.

RAO See Read-As-One (RAO).

RAZ See Read-As-Zero (RAZ).

RAO/SBOP In versions of the Arm architecture before Armv8, Read-As-One, Should-Be-One-or-Preserved on writes.

From the introduction of the Armv8 architecture, RES1 replaces this description.

See also UNK/SBOP, Read-As-One (RAO), RES1, Should-Be-One-or-Preserved (SBOP).

RAO/WI Read-As-One, Writes Ignored.

Hardware must implement the field as Read-As-One, and must ignore writes to the field.

Software can rely on the field reading as all 1s, and on writes being ignored.

This description can apply to a single bit that reads as 1, or to a field that reads as all 1s.

See also Read-As-One (RAO).

RAZ/SBZP In versions of the Arm architecture before Armv8, Read-As-Zero, Should-Be-Zero-or-Preserved on writes.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

From the introduction of the Armv8 architecture, RES0 replaces this description.

See also UNK/SBZP, Read-As-Zero (RAZ), RES0, Should-Be-Zero-or-Preserved (SBZP).

RAZ/WI Read-As-Zero, Writes Ignored.

Hardware must implement the field as Read-As-Zero, and must ignore writes to the field.

Software can rely on the field reading as all 0s, and on writes being ignored.

This description can apply to a single bit that reads as 0, or to a field that reads as all 0s.

See also Read-As-Zero (RAZ).

Read-allocate cache A cache in which a cache miss on reading data causes a cache line to be allocated into the cache.

Read-As-One (RAO) Hardware must implement the field as reading as all 1s.

Software:

- Can rely on the field reading as all 1s.
- Must use a SBOP policy to write to the field.

This description can apply to a single bit that reads as 1, or to a field that reads as all 1s.

See also RAO/SBOP, RAO/WI, RES1.

Read-As-Zero (RAZ) Hardware must implement the field as reading as all 0s.

Software:

- Can rely on the field reading as all 0s
- Must use a SBZP policy to write to the field.

This description can apply to a single bit that reads as 0, or to a field that reads as all 0s.

See also RAZ/SBZP, RAZ/WI, RES0.

Read-to-Clear (RC) A read of the field clears the field to 0 once the read is complete.

Read, modify, write In a read, modify, write instruction sequence, a value is read to a general-purpose register, the relevant fields updated in that register, and the new value written back.

Recoverable error A contained error that must be corrected to allow the correct operation of the system or smaller parts of the system to continue.

Reliability Continuity of correct service.

Requester An agent in a computing system that is capable of initiating memory transactions.

See also Completer.

RES0 A reserved bit. Used for fields in register descriptions, and for fields in architecturally-defined data structures that are held in memory, for example in translation table descriptors.

Within the architecture, there are some cases where a register bit or field:

- Is RES0 in some defined architectural context.
- Has different defined behavior in a different architectural context.

This means the definition of RES0 for fields in read/write registers is:

If a bit is RES0 in all contexts

For a bit in a read/write register, it is IMPLEMENTATION DEFINED whether:

1. The bit is hardwired to 0. In this case:
   - Reads of the bit always return 0.
   - Writes to the bit are ignored.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2. The bit can be written. In this case:
    - An indirect write to the register sets the bit to 0.
    - A read of the bit returns the last value successfully written, by either a direct or an indirect write, to the bit.

      If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.
    - A direct write to the bit must update a storage location associated with the bit.
    - The value of the bit must have no effect on the operation of the PE, other than determining the value read back from the bit, unless this Manual explicitly defines additional properties for the bit.

Whether RES0 bits or fields follow behavior 1 or behavior 2 is IMPLEMENTATION DEFINED on a bit-by-bit basis.

If a bit is RES0 only in some contexts

For a bit in a read/write register, when the bit is described as RES0:

- An indirect write to the register sets the bit to 0.
- A read of the bit must return the value last successfully written to the bit, by either a direct or an indirect write, regardless of the use of the register when the bit was written.

  If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.
- A direct write to the bit must update a storage location associated with the bit.
- While the use of the register is such that the bit is described as RES0, the value of the bit must have no effect on the operation of the PE, other than determining the value read back from that bit, unless this Manual explicitly defines additional properties for the bit.

Considering only contexts that apply to a particular implementation, if there is a context in which a bit is defined as RES0, another context in which the same bit is defined as RES1, and no context in which the bit is defined as a functional bit, then it is IMPLEMENTATION DEFINED whether:

- Writes to the bit are ignored, and reads of the bit return an UNKNOWN value.
- The value of the bit can be written, and a read returns the last value written to the bit.

The RES0 description can apply to bits or fields that are read-only, or are write-only:

- For a read-only bit, RES0 indicates that the bit reads as 0, but software must treat the bit as UNKNOWN.
- For a write-only bit, RES0 indicates that software must treat the bit as SBZ.

A bit that is RES0 in a context is reserved for possible future use in that context. To preserve forward compatibility, software:

- Must not rely on the bit reading as 0.
- Must use an SBZP policy to write to the bit.

This RES0 description can apply to a single bit, or to a field for which each bit of the field must be treated as RES0.

In body text, the term RES0 is shown in SMALL CAPITALS.

See also Read-As-Zero (RAZ), RES0H, RES1, Should-Be-Zero-or-Preserved (SBZP), UNKNOWN.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RES0H A reserved bit or field with SBZP. This behavior uses the hardwired to 0 subset of the RES0 definition.

See also Read-As-Zero (RAZ), RES0, RES1, Should-Be-Zero-or-Preserved (SBZP).

RES1 A reserved bit. Used for fields in register descriptions, and for fields in architecturally-defined data structures that are held in memory, for example in translation table descriptors.

Within the architecture, there are some cases where a register bit or field:

- Is RES1 in some defined architectural context.
- Has different defined behavior in a different architectural context.

This means the definition of RES1 for fields in read/write registers is:

If a bit is RES1 in all contexts

For a bit in a read/write register, it is IMPLEMENTATION DEFINED whether:

1. The bit is hardwired to 1. In this case:
   - Reads of the bit always return 1.
   - Writes to the bit are ignored.
2. The bit can be written. In this case:
   - An indirect write to the register sets the bit to 1.
   - A read of the bit returns the last value successfully written, by either a direct or an indirect write, to the bit.

     If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.
   - A direct write to the bit must update a storage location associated with the bit.
   - The value of the bit must have no effect on the operation of the PE, other than determining the value read back from the bit, unless this Manual explicitly defines additional properties for the bit.

Whether RES1 bits or fields follow behavior 1 or behavior 2 is IMPLEMENTATION DEFINED on a bit-by-bit basis.

If a bit is RES1 only in some contexts

For a bit in a read/write register, when the bit is described as RES1:

- An indirect write to the register sets the bit to 1.
- A read of the bit must return the value last successfully written to the bit, regardless of the use of the register when the bit was written.

  Note

  As indicated in this list, this value might be written by an indirect write to the register.

  If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.
- A direct write to the bit must update a storage location associated with the bit.
- While the use of the register is such that the bit is described as RES1, the value of the bit must have no effect on the operation of the PE, other than determining the value read back from that bit, unless this Manual explicitly defines additional properties for the bit.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Considering only contexts that apply to a particular implementation, if there is a context in which a bit is defined as RES0, another context in which the same bit is defined as RES1, and no context in which the bit is defined as a functional bit, then it is IMPLEMENTATION DEFINED whether:

- Writes to the bit are ignored, and reads of the bit return an UNKNOWN value.
- The value of the bit can be written, and a read returns the last value written to the bit.

The RES1 description can apply to bits or fields that are read-only, or are write-only:

- For a read-only bit, RES1 indicates that the bit reads as 1, but software must treat the bit as UNKNOWN.
- For a write-only bit, RES1 indicates that software must treat the bit as SBO.

A bit that is RES1 in a context is reserved for possible future use in that context. To preserve forward compatibility, software:

- Must not rely on the bit reading as 1.
- Must use an SBOP policy to write to the bit.

This RES1 description can apply to a single bit, or to a field for which each bit of the field must be treated as RES1.

In body text, the term RES1 is shown in SMALL CAPITALS.

See also Read-As-One (RAO), RES0, RES0H, Should-Be-One-or-Preserved (SBOP), UNKNOWN.

Reserved Unless otherwise stated:

- Instructions that are reserved or that access reserved registers have UNPREDICTABLE or CONSTRAINED UNPREDICTABLE behavior.
- Bit positions described as reserved are:
  - In an RW or WO register, RES0.
  - In an RO register, UNK.

See also CONSTRAINED UNPREDICTABLE, RES0, RES0H, RES1, UNDEFINED, UNK, UNPREDICTABLE.

Resource partition The collection of MPAM resource control settings associated with a software environment and identified by the combination of a physical PARTID space and a partition number.

RESS Reserved, Sign extended. A register value is extended by copying the sign bit into all of the reserved bits to the left of the most significant bit of the field. The values of these bits are identical to the most significant bit of the value being extended.

Within the architecture, a register bit or field can be treated:

- As RESS in few defined architectural contexts.
- In a different defined behavior in other architectural contexts.

This means the definition of RESS for fields in read/write registers is:

If a field [Q:P] is RESS in all contexts

For a bit in a read/write register, it is IMPLEMENTATION DEFINED whether:

1. The bit is hardwired as a sign-extension of bit [P-1]. In this case:
   - Reads of the bit always return the value of bit [P-1].
   - Writes to the bit are ignored.
2. The bit can be written. In this case:
   - An indirect write to the register sets the bit to the same value as bit [P-1].

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- A read of the bit returns the last value successfully written, by either a direct or an indirect write, to the bit.

  If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.

- A direct write to the bit must update a storage location associated with the bit.
- The value of the bit must have no effect on the operation of the PE, other than determining the value read back from the bit, unless this Manual explicitly defines additional properties for the bit.

Whether RESS bits or fields follow behavior 1 or behavior 2 is IMPLEMENTATION DEFINED on a bit-by-bit basis.

If a field [Q:P] is RESS only in some contexts

For a bit in a read/write register, when the bit is described as RESS:

- An indirect write to the register sets the bit to the same value as bit [P-1].
- A read of the bit must return the value last successfully written to the bit, regardless of the use of the register when the bit was written.

  Note

  As indicated in this list, this value might be written by an indirect write to the register.

  If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.
- A direct write to the bit must update a storage location associated with the bit.
- While the use of the register is such that the bit is described as RESS, the value of the bit must have no effect on the operation of the PE, other than determining the value read back from that bit, unless this Manual explicitly defines additional properties for the bit.

Considering only contexts that apply to a particular implementation, if there is a context in which a bit is defined as RES0, another context in which the same bit is defined as RES1, and no context in which the bit is defined as a functional bit, then it is IMPLEMENTATION DEFINED whether:

- Writes to the bit are ignored, and reads of the bit return an UNKNOWN value.
- The value of the bit can be written, and a read returns the last value written to the bit.

The RESS description can apply to bits or fields that are read-only, or are write-only:

- For a read-only bit, RESS indicates that the bit reads as the value of bit [P-1], but software must treat the bit as UNKNOWN.
- For a write-only bit, RESS indicates that software must treat the bit as SBO or SBZ as necessary.

A bit that is RESS in a context is reserved for possible future use in that context. To preserve forward compatibility, software:

- Must not rely on the bit reading as [P-1].
- Must use a SBOP or SBZP policy to write to the bit.

This RESS description can apply to a single bit, or to a field for which each bit of the field must be treated as RESS.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Rewind point A rewind point is a point in the program flow to which execution can return if all subsequent execution is found to have been incorrectly speculatively executed.

RISC Reduced Instruction Set Computer.

Rounding error The value of the rounded result of an arithmetic operation minus the exact result of the operation.

Rounding mode Specifies how the exact result of a floating-point operation is rounded to a value that is representable in the destination format. The rounding modes are defined by the *IEEE Standard for Floating-point Arithmetic*, see Floating-point standards, and terminology.

Saturated arithmetic Integer arithmetic in which a result that would be greater than the largest representable number is set to the largest representable number, and a result that would be less than the smallest representable number is set to the smallest representable number. Signed saturated arithmetic is often used in DSP algorithms. It contrasts with the normal signed integer arithmetic used in Arm processors, in which overflowing results wrap around from $+2^{31}$-1 to -2^{31} or vice versa.

SBO See Should-Be-One (SBO).

SBOP See Should-Be-One-or-Preserved (SBOP).

SBZ See Should-Be-Zero (SBZ).

SBZP See Should-Be-Zero-or-Preserved (SBZP).

Scalar base register A scalar base register refers to an AArch64 general-purpose register, X0-X31, or the current stack pointer, SP.

Scalar index register A scalar index register refers to an AArch64 general-purpose register, X0-X31, or for certain instructions, XZR.

SDC See Silent Data Corruption (SDC).

SDC FIT rate The FIT rate for failures because of SDC.

SDEC Single device error correction EDAC. This can detect and correct multiple clustered errors in a protection granule, such as the types of errors that might be seen if a protection granule is striped across multiple devices and multiple errors come from a single device.

SECDED Single error correct, double error detect EDAC. This can detect a single or double bit error and correct a single bit error in a protection granule.

SED Single error detect EDC. This can detect a single bit error in a protection granule.

Security hole A mechanism by which execution at the current level of privilege can achieve an outcome that cannot be achieved at the current or a lower level of privilege using instructions that are not UNPREDICTABLE and are not CONSTRAINED UNPREDICTABLE. The Arm architecture forbids security holes.

See also CONSTRAINED UNPREDICTABLE, UNPREDICTABLE.

Self-modifying code Code that writes one or more instructions to memory and then executes them. When using self-modifying code, you must use cache maintenance and barrier instructions to ensure synchronization. For more information, see Caches and memory hierarchy.

SError exception An asynchronous exception in the Arm architecture.

Serviceability The ability to undergo modifications and repairs.

Service failure mode A mode entered to reduce the severity of an error.

Set See Cache sets.

Should-Be-One (SBO) Hardware must ignore writes to the field.

Arm strongly recommends that software writes the field as all 1s. If software writes a value that is not all 1s, it must expect an UNPREDICTABLE or CONSTRAINED UNPREDICTABLE result.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This description can apply to a single bit that should be written as 1, or to a field that should be written as all 1s.

See also CONSTRAINED UNPREDICTABLE, UNPREDICTABLE.

Should-Be-One-or-Preserved (SBOP) From the introduction of the Armv8 architecture, the description Should-Be-One-or-Preserved (SBOP) is superseded by RES1.

Hardware must ignore writes to the field.

When writing this field, software must either write all 1s to this field or, if the register is being restored from a previously read state, write the previously read value to this field. If this is not done, then the result is unpredictable.

This description can apply to a single bit that should be written as its preserved value or as 1, or to a field that should be written as its preserved value or as all 1s.

See also CONSTRAINED UNPREDICTABLE, UNPREDICTABLE.

Should-Be-Zero (SBZ) Hardware must ignore writes to the field.

Arm strongly recommends that software writes the field as all 0s. If software writes a value that is not all 0s, it must expect an UNPREDICTABLE or CONSTRAINED UNPREDICTABLE result.

This description can apply to a single bit that should be written as 0, or to a field that should be written as all 0s.

See also CONSTRAINED UNPREDICTABLE, UNPREDICTABLE.

Should-Be-Zero-or-Preserved (SBZP) From the introduction of the Armv8 architecture, the description Should-Be-Zero-or-Preserved (SBZP) is superseded by RES0.

Hardware must ignore writes to the field.

When writing this field, software must either write all 0s to this field or, if the register is being restored from a previously read state, write the previously read value to this field. If this is not done, then the result is unpredictable.

This description can apply to a single bit that should be written as its preserved value or as 0, or to a field that should be written as its preserved value or as all 0s.

See also CONSTRAINED UNPREDICTABLE, UNPREDICTABLE.

Signaling NaN An Invalid Operation floating-point exception occurs whenever any floating-point operation receives a signaling NaN as an operand. Signaling NaNs can be used in debugging, to track down some uses of uninitialized variables.

See also NaN, Quiet NaN.

Signed immediate and offset fields Are encoded in two's complement notation unless otherwise stated.

Silent Data Corruption (SDC) An error that is not detected by hardware or software.

SIMD Single-Instruction, Multiple-Data.

Simple sequential execution The behavior of an implementation that fetches, decodes and completely executes each instruction before proceeding to the next instruction. Such an implementation performs no speculative accesses to memory, including to instruction memory. The implementation does not pipeline any phase of execution. In practice, this is the theoretical execution model that the architecture is based on, and Arm does not expect this model to correspond to a realistic implementation of the architecture.

Single-copy atomicity The form of atomicity described in Properties of single-copy atomic accesses.

See also Atomicity, Multi-copy atomicity.

Single-precision value A 32-bit word that is interpreted as a basic single-precision floating-point number according to the *IEEE Standard for Floating-point Arithmetic*.

SMMU System Memory Management Unit.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Spatial locality The observed effect that after a program has accessed a memory location, it is likely to also access nearby memory locations in the near future. Caches with multi-word cache lines exploit this effect to improve performance.

Special-purpose register One of the Special purpose registers for which all direct and indirect reads and writes to the register appear to occur in program order relative to other instructions, without the need for any explicit synchronization.

Speculative Speculative operations are:

- Operations that are generated by instructions that appear in the Execution stream after a branch that is not architecturally resolved.
- Operations that are generated by instructions that appear in the Execution stream after an instruction where a synchronous exception condition has not been architecturally resolved.
- Operations that are generated by conditional instructions for which the conditions for the instruction have not been architecturally resolved.
- Operations that are generated by instructions that appear in the Execution stream after the point at which a precise asynchronous exception will be taken.
- Operations generated by instructions for which one or more of the arguments come from a register that has not been architecturally resolved.
- Operations generated by the hardware that are not directly generated by any instructions appearing in the Execution stream.
- Memory effects (M2) generated by load or store instructions (LS2) appearing in program order after load or store instructions (LS1) that generate memory effects (M1) where all of the following apply:
  - M1 is locally-ordered-before M2.
  - LS1 has not been executed before LS2.
- Read accesses generated for a translation table walk for which the granule protection check for the address being accessed has not been architecturally resolved.

See also Execution stream.

SPI Shared Peripheral Interrupt.

Superpriority An attribute that is used to denote virtual and physical IRQ and FIQ interrupts as non-maskable. IRQ and FIQ interrupts with Superpriority can be taken under certain conditions where usually they would be masked by the PSTATE.{I, F} bits.

System Control Processor A PE dedicated to system control and monitoring.

Temporal locality The observed effect that after a program has accesses a memory location, it is likely to access the same memory location again in the near future. Caches exploit this effect to improve performance.

TLB See Translation Lookaside Buffer (TLB).

TLB lockdown A way to prevent specific translation table entries from being evicted from a TLB. This ensures that accesses to the associated regions of virtual memory never cause a translation table walk.

Translation Lookaside Buffer (TLB) A microarchitectural structure containing the results of translation table walks. They help to reduce the average cost of a memory access.

Translation table A table held in memory that defines the properties of regions of virtual memory.

Translation table walk The process of doing a full translation table lookup. It is performed automatically by hardware.

Trap enable bits In VFPv2, VFPv3U, and VFPv4U, determine whether trapped or untrapped exception handling is selected. If trapped exception handling is selected, the way it is carried out is IMPLEMENTATION DEFINED.

Unaligned An unaligned access is an access where the address of the access is not aligned to the size of an element of the access.

Unaligned memory accesses Are memory accesses that are not, or might not be, appropriately halfword-aligned, word-aligned, or doubleword-aligned.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Unallocated Except where otherwise stated in this manual, an instruction encoding is unallocated if the architecture does not assign a specific function to the entire bit pattern of the instruction, but instead describes it as CONSTRAINED UNPREDICTABLE, UNDEFINED, UNPREDICTABLE, or as an unallocated hint instruction.

A bit in a register is unallocated if the architecture does not assign a function to that bit.

See also CONSTRAINED UNPREDICTABLE, UNDEFINED, UNPREDICTABLE.

Uncontainable error See Uncontained error.

Uncontained error An error that has been, or might have been, silently propagated.

Uncorrected error An error caused by a fault which was not removed.

UNDEFINED Indicates cases where an attempt to execute a particular encoding bit pattern generates an exception, that is taken to the current Exception level, or to the default Exception level for taking exceptions if the UNDEFINED encoding was executed at EL0. This applies to:

- Any encoding that is not allocated to any instruction.
- Any encoding that is defined as never accessible at the current Exception level.
- Some cases where an enable, disable, or trap control means an encoding is not accessible at the current Exception level.

Note

On reset, the default Exception level for taking exceptions from EL0 is EL1. However, an implementation might include controls that can change this, effectively making EL1 inactive. See the description of the Exception model for more information

In body text, the term UNDEFINED is shown in SMALL CAPITALS.

See also Undefined Instruction exception.

Undetected error See Latent error.

Unified cache Is a cache used for both processing instruction fetches and processing data loads and stores.

Unindexed addressing Means addressing in which the base register value is used directly as the virtual address to send to memory, without adding or subtracting an offset. In most types of load/store instruction, unindexed addressing is performed by using offset addressing with an immediate offset of 0.

UNK An abbreviation indicating that software must treat a field as containing an UNKNOWN value.

Hardware must implement the bit as read as 0, or all 0s for a multi-bit field. Software must not rely on the field reading as zero.

See also UNKNOWN.

UNK/SBOP Hardware must implement the field as Read-As-One, and must ignore writes to the field.

Software must not rely on the field reading as all 1s, and except for writing back to the register it must treat the value as if it is UNKNOWN. Software must use an SBOP policy to write to the field.

This description can apply to a single bit that should be written as its preserved value or as 1, or to a field that should be written as its preserved value or as all 1s.

See also Read-As-One (RAO), Should-Be-One-or-Preserved (SBOP), UNKNOWN.

UNK/SBZP Hardware must implement the bit as Read-As-Zero, and must ignore writes to the field.

Software must not rely on the field reading as all 0s, and except for writing back to the register must treat the value as if it is UNKNOWN. Software must use an SBZP policy to write to the field.

This description can apply to a single bit that should be written as its preserved value or as 0, or to a field that should be written as its preserved value or as all 0s.

See also Read-As-Zero (RAZ), Should-Be-Zero-or-Preserved (SBZP), UNKNOWN.

UNKNOWN An UNKNOWN value does not contain valid data, and can vary from implementation to implementation. An UNKNOWN value must not return information that cannot be accessed at the current or a lower level

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

of privilege using instructions that are not UNPREDICTABLE, are not CONSTRAINED UNPREDICTABLE, and do not return UNKNOWN values.

An UNKNOWN value can vary from moment to moment, and instruction to instruction, unless it has previously been assigned, other than at reset, to one of the following registers:

- Any of the general-purpose registers.
- Any of the Advanced SIMD and floating-point registers.
- Any of the Scalable Vector Extension registers.
- Any of the PSTATE N, Z, C, or V flags.

An UNKNOWN value must not be documented or promoted as having a defined value or effect.

In body text, the term UNKNOWN is shown in SMALL CAPITALS.

See also CONSTRAINED UNPREDICTABLE, UNDEFINED, UNK, UNPREDICTABLE.

UNPREDICTABLE Means the behavior cannot be relied upon. UNPREDICTABLE behavior must not perform any function that cannot be performed at the current or a lower level of privilege using instructions that are not UNPREDICTABLE.

UNPREDICTABLE behavior must not be documented or promoted as having a defined effect.

An instruction that is UNPREDICTABLE can be implemented as UNDEFINED.

Execution at Non-secure EL1 or EL0 of an instruction that is UNPREDICTABLE can be implemented as generating a trap exception that is taken to EL2, provided that at least one instruction that is not UNPREDICTABLE and is not CONSTRAINED UNPREDICTABLE causes a trap exception that is taken to EL2.

In body text, the term UNPREDICTABLE is shown in SMALL CAPITALS.

See also CONSTRAINED UNPREDICTABLE, UNDEFINED.

Unrecoverable error A contained error that is not recoverable. Continued correct operation is generally not possible. Usually this means correct operation of the system, but it can also be used in other contexts to describe correct operation of a smaller part. Systems might use high-level recovery techniques to work around an unrecoverable yet contained error in a component so that the system recovers from the error.

Upstream Information propagating in the direction from terminating Completer components towards Requesters.

VA See Virtual address (VA).

VFP In Armv7, an extension to the Arm architecture, that provides single-precision and double-precision floating-point arithmetic.

Vector A one-dimensional array of vector elements of the same size and data type.

Vector element Individual subdivisions of a vector register that can be 8, 16, 32, 64 or 128 bits in size. The vector element size and data type is specified independently by each instruction.

Virtual address (VA) An address generated by an Arm PE. This means it is an address that might be held in the program counter of the PE. For a PMSA implementation, the virtual address is identical to the physical address.

See also Intermediate physical address (IPA), Physical address (PA).

VM Virtual Machine.

VMM Virtual Machine Monitor. An alias for “hypervisor”.

Virtual PARTID A PARTID in MPAM that can be used by a virtual machine (VM). Virtual PARTIDs are mapped into physical PARTIDs using the virtual partition mapping entries in the MPAMVPM0 - MPAMVPM7 registers.

Watchpoint A debug event triggered by an access to memory, specified in terms of the address of the location in memory being accessed.

Way See Cache way.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

WI Writes Ignored. In a register that software can write to, a WI attribute applied to a bit or field indicates that the bit or field ignores the value written by software and retains the value it had before that write.

See also RAO/WI, RAZ/WI, RES0, RES1.

Word A 32-bit data item. Words are normally word-aligned in Arm systems.

Word-aligned Means that the address is divisible by 4.

Write-allocate cache A cache in which a cache miss on storing data causes a cache line to be allocated into the cache.

Write-back cache A cache in which when a cache hit occurs on a write access, the data is only written to the cache. Data in the cache can therefore be more up-to-date than data in the next level of cache or memory. Any such data is written back to the next level of cache or memory when the cache line is cleaned. Another common term for a write-back cache is a *copy-back cache*.

Write-One-to-Clear (W1C) Writes of 0 to the bit are ignored. A write of 1 clears the bit to 0.

Write-One-to-Set (W1S) Writes of 0 to the bit are ignored. A write of 1 sets the bit to 1.

Write-Through cache A cache in which when a cache hit occurs on a write access, the data is written both to the cache and to the next level of cache or memory.

Write buffer A block of high-speed memory that optimizes stores to main memory.

Copyright © 2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential